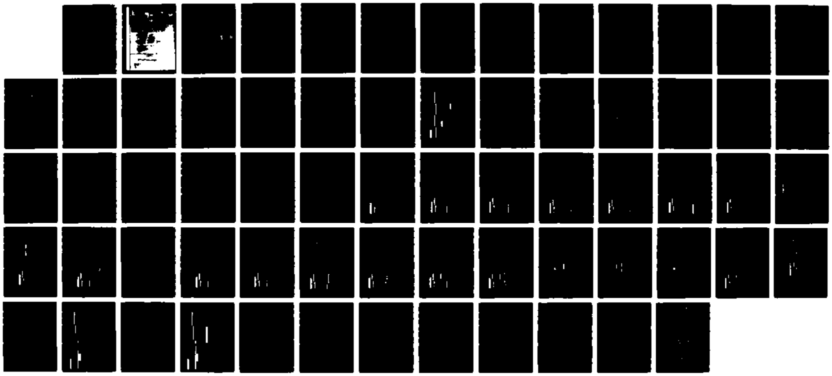
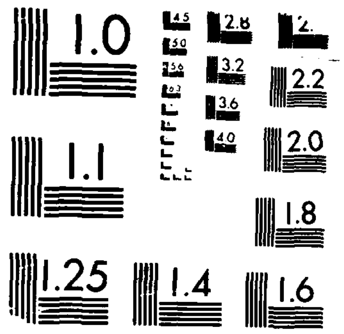
AD-A193 702   OBJECT-ORIENTED SIMULATION OF EW SYSTEMS(U) DEFENCE RESEARCH ESTABLISHMENT OTTAWA (ONTARIO) B M BARRY DEC 87 DREO-TN-87-31   1/1

UNCLASSIFIED   F/G 17/4   NL





MICROCOPY RESOLUTION TEST CHART

[illegible]REAU [illegible] STANDARDS 1963-A

1

National Defence / Défense nationale

DEFENSIO PER SCIENTIAM

OBJECT-ORIENTED SIMULATION OF EW SYSTEMS

by

Brian M. Barry

DEFENCE RESEARCH ESTABLISHMENT OTTAWA

TECHNICAL NOTE 87-31

Canada

December 1987
Ottawa

1

National Defence Défense nationale

OBJECT-ORIENTED SIMULATION OF EW SYSTEMS

by

Brian M. Barry
Radar ESM Section
Electronic Warfare Division

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
TECHNICAL NOTE 87-31

PCN
011LB13

December 1987
Ottawa

ACKNOWLEDGEMENT

The author wishes to acknowledge his debt to the many colleagues who have engaged in stimulating and thought provoking discussions about simulations and validation over the years, including Alan Sewards, Dave Thomas, Fred Symons, Stan Leroy, Leo Makela and David Jones. Special thanks are due to Allen Brown, who through his painstaking efforts as editor significantly improved the presentation of this material.

ABSTRACT

Simulations of complex EW systems are difficult to build and virtually impossible to thoroughly validate. As a consequence, most EW systems engineers tend to regard results derived from simulations as suspect, preferring to rely instead on laboratory testing and field trials for performance evaluations. We suggest that the real problem may be that traditional simulations do not provide the kind of modelling and analysis tools which the systems engineer really needs. In this paper a prototype for a new kind of EW simulation environment which supports an object-oriented approach to modelling and simulation is described. We will provide some background information on object-oriented programming, describe the software architecture of the simulation environment, and discuss several examples which illustrate its use.

RESUME

La simulation de systèmes complexes de guerre électronique est difficile à mettre en oeuvre et pratiquement impossible à vérifier. Par conséquent ceux qui pratiquent le génie des systèmes ont tendance a en considerer les résultats comme douteux, préférant en général estimer la performance de ces systèmes par des mesures en laboratoire ou en champs d'essais. Il est suggéré ici que le problème réel est du au fait que les méthodes de simulation traditionelles n'offrent pas le genre d'outils necessaires à la modélisation et à l'analyse dont le génie des systèmes a réellement besoin. Le prototype d'un nouveau genre d'environnement spécialisé pour la simulation de systèmes de guerre électronique et basé sur une configuration du logiciel a structure d'objets est décrit dans se document. De l'information générale sur la programmation en structure d'object est offerte ainsi qu'une description de l'architecture du logiciel utilisé pour le simulation. Un certain nombre d'exemples servant à illustré l'ulilisation du nouvel environnement sont aussi discutés.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (cont)

LIST OF TABLES

1.0 INTRODUCTION

EW technology and techniques are constantly evolving in response to a changing signal environment. As EW systems continue to grow in complexity, systems analysts and design engineers are increasingly using simulations to understand and predict system performance. Unfortunately, simulation technology has not been able to keep pace with the advances which have been made in EW systems technology. It is commonplace for simulations of large-scale systems to exhibit problems of intelligibility, modifiability, credibility, validity, and performance.

We distinguish between a conceptual or abstract model of some real-world phenomenon, and a digital simulation of that phenomenon (i.e. a computer program which realizes the abstract model). By verification we mean the process of demonstrating that the simulation is a true and correct realization of the abstract model. Validation is the process of proving that both the model and its realization, the simulation, adequately represent those aspects of the phenomenon under study. Clearly the definition of "adequate" depends on the intended application of the simulation. Note also that we have been careful to use the phrase "adequately represent" rather than "correctly represent": absolute validity is obviously an unreachable goal. Credibility refers to the users' belief that the simulation will be a good predictor of real-world behaviour. Credibility is an attribute which is gradually acquired by a simulation. One hopes that credibility will be an end-product of the validation and verification processes. These ideas are summarized in Figure 1.

In practice, most EW systems engineers tend to regard results derived from simulations as suspect, preferring to rely instead on laboratory testing and field trials for performance evaluations. In this paper we will suggest some underlying explanations for these credibility problems based on our own experience in developing and using simulations of EW systems. A central issue is that, too often, validation is an afterthought, whereas it should be a design goal. To remedy this, we propose that more effort should be expended on developing software tools which directly support the design of simulations. In recent years it has become clear that programming support environments can make software engineers more productive and at the same time improve software reliability. We will argue (hopefully persuasively!) that the utility of EW simulations could be significantly increased if analogous simulation support environments were developed specifically for EW applications. In our discussion, we will try to identify the capabilities which we believe such a programming environment should have, and we will describe our approach to implementation.

The organization of this paper is straightforward. In Section 2, the shortcomings of current simulation methodologies are examined. Based on this analysis, some of the features which one would like to see in a programming environment specifically designed for developing EW simulations will be discussed. Section 3 will describe a prototype of an object-oriented EW simulation environment written in the Smalltalk language. This prototype is intended to be a "proof of concept" system which would demonstrate the feasibility of providing many of the desirable features described in Section 2. In Section 4 we will evaluate the prototype in the light of these requirements, and describe several enhancements which could improve its performance.

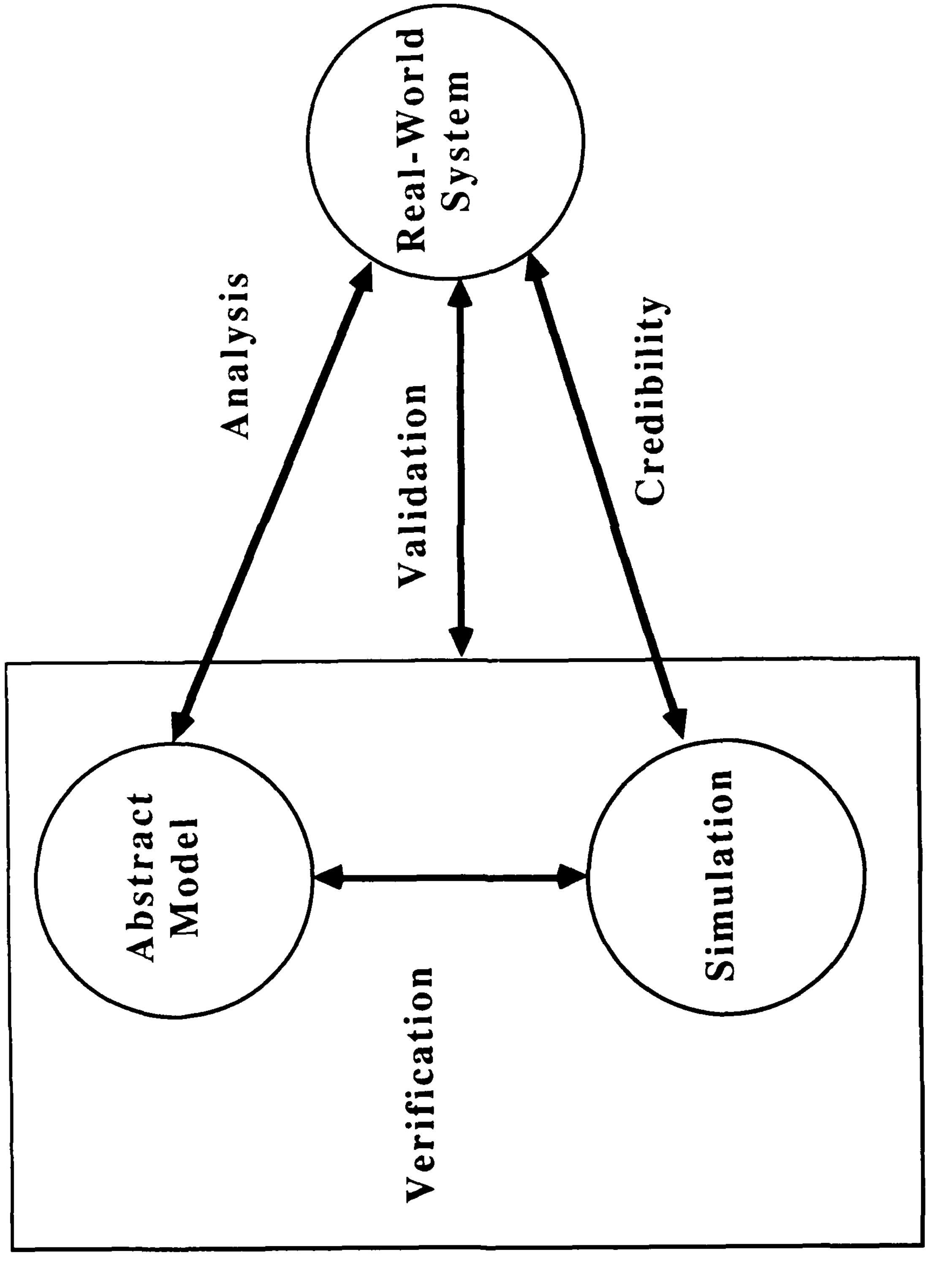


FIGURE 1: Relationships Between Models, Simulations, and Reality

2.0 BACKGROUND: THE NEED FOR NEW APPROACHES

Our effort to design and build an EW simulation environment is part of an on-going project aimed at developing a new signal processor for ESM applications [1]. This new system architecture will employ a loosely coupled network of multiprocessor computers (Figure 2), each of which will consist of a filter subsystem and a number of single board computers with associated memory (Figure 3). Incoming data will be divided into parallel streams by microprocessor controlled hardware filters. These filter subsystems will be implemented using highly parallel VLSI modules; filtered data will be stored in multiple circular buffers which are directly accessible by the single board computers at a high level. The single board computers will be run in a tightly coupled fashion under a message-passing operating system. Signal processing algorithms and control software will be implemented in a highly modular, task-oriented style. An integrated testbed system is being developed to implement and investigate this design with a combination of "breadboards" and commercially available components. This testbed will include a vertically integrated development and test environment which will provide software support for program generation, simulation, and performance monitoring.

The simulation component of this development environment is intended to be used for both design (during system development) and analysis (for evaluating the effectiveness of the new equipment). When planning this activity, it became clear that to address these requirements a flexible system with rapid response time was needed. During this project we anticipate that a number of increasingly complex versions of the basic system design will be developed. Consequently, in order to adequately support development work, the simulation environment would have to concurrently support a number of models constructed with varying levels of detail; this support ideally would include a version management capability which makes the underlying relationships between the models explicit. To make the best use of available manpower we also wanted an articulate system with a natural user interface. It was our philosophy that the simulation had to be accessible by all project engineers, not just a few "simulation gurus". Above all, we needed to produce believable results which could be validated.

2.1 Why Do Conventional Simulations Lack Credibility?

Very early in the project we decided to develop a new software package specifically designed for simulating EW systems rather than try to adapt an existing simulation to our application. The two related issues of credibility and validity were our main considerations when making this decision. As scientists and engineers, we build models of systems as an aid to understanding their function. We need models in order to reason about these systems. By subjecting our models to test and scrutiny, we are able to refine and improve them; in other words, we learn. Simulations enter the picture because a detailed model of a complex device (such as an EW system) is frequently too involved for mental manipulation. We require a computational aid, such as a computer, to fully exploit and exercise the model.

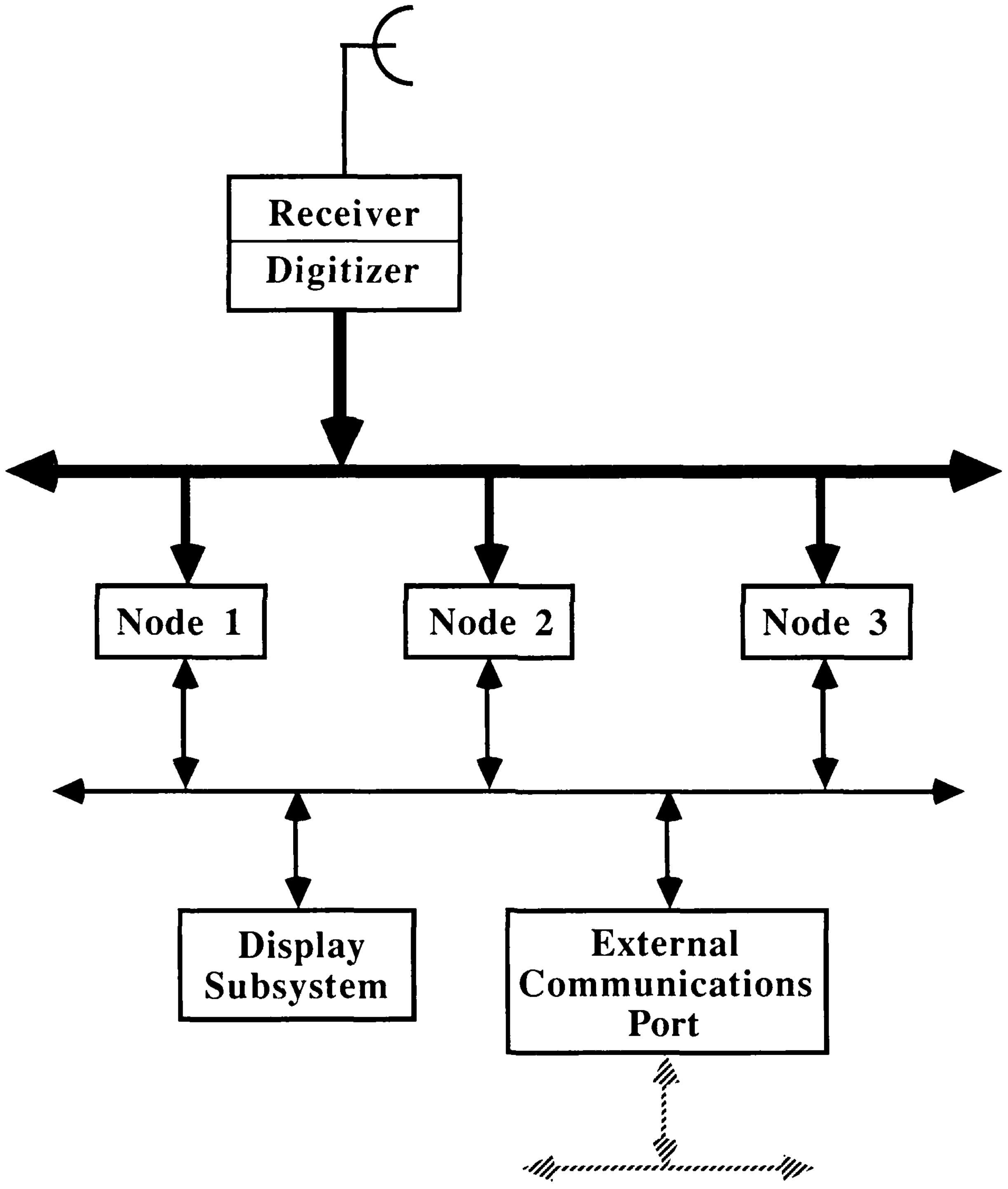


FIGURE 2: Multiprocessor System Architecture

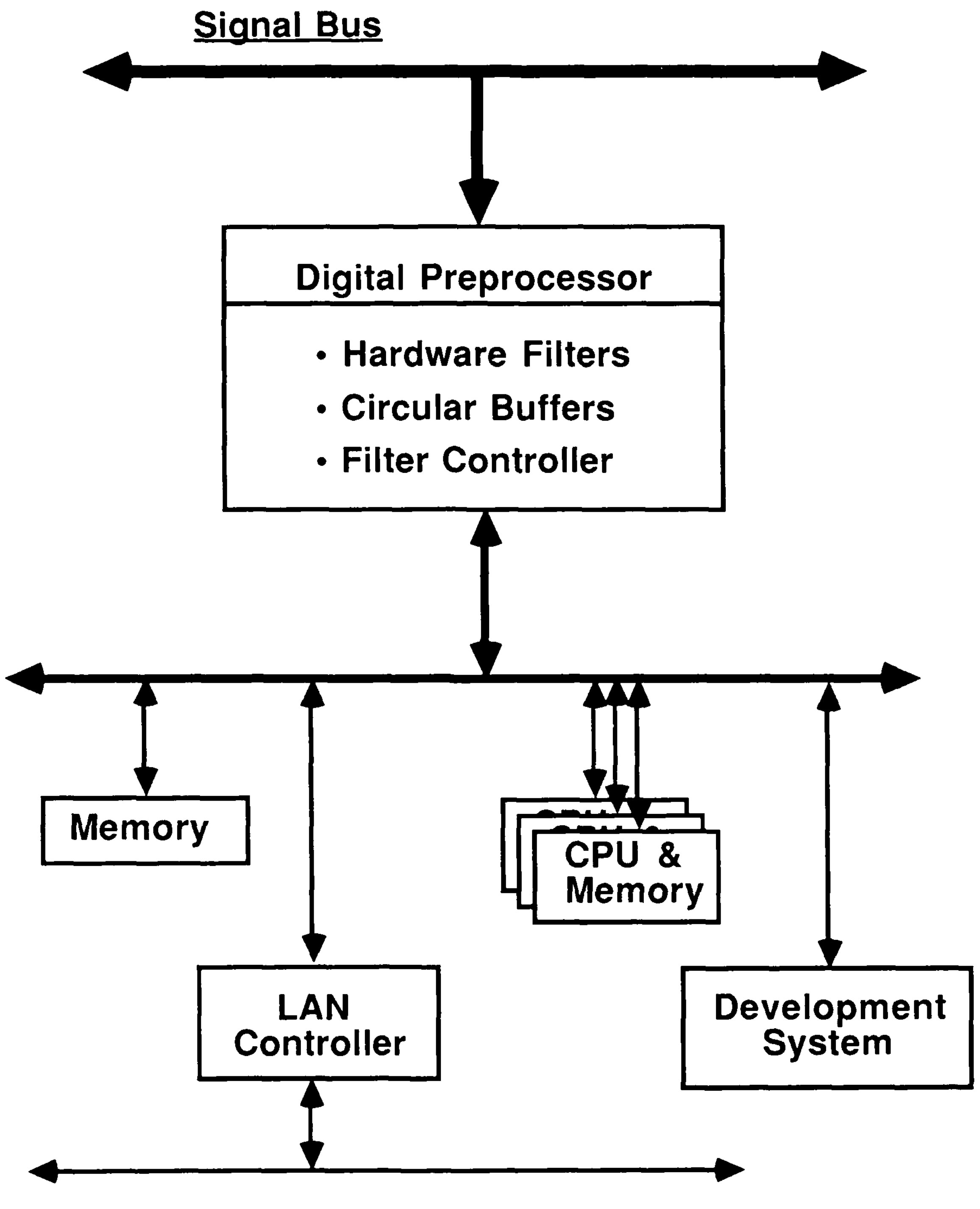


FIGURE 3: Single Node Configuration

Unfortunately, traditional simulation approaches do not support the scientific method (i.e. the "learning by modelling" paradigm) very well. In the typical case the EW engineering group probably includes a team of Fortran or Pascal programmers whose only function is to implement models. Moreover, the EW engineer usually must wait weeks, or even months, to see the results (which almost always are not what he expected!). Consequently, the results derived from the simulation frequently come much to late to have any impact on decision makers. The whole process seems to lack credibility: given a conflict between the simulated results and the engineer's own intuition, intuition almost always wins.

Credibility problems are also closely related to the absence of any systematic methodology for validating the results of simulations. Currently, the usual approach to validation involves exercising the software until a consensus has formed among the users that most of the serious deficiencies have been detected and corrected. At this point, performance tests, designed on an ad hoc basis (often based on real test data if it is available), will be carried out to gain increased confidence in the quality of the model. Our own experience leads us to conclude that considerable effort is required to achieve even a moderate level of credibility by this route. Moreover, in many cases the system under development has itself already been built and tested before the simulation has been satisfactorily validated. One is often led to wonder what benefit (if any!) the simulation provided the system developers.

2.2 Why is Validation so Time-Intensive?

We believe that expert review, rather than pseudo-formalized ad hoc testing by the simulation designers, is the only truly viable way to establish the validity of a simulation model. Indeed, we would suggest that understanding and credibility are synonymous. For a complex simulation, true understanding can be very costly indeed!

Unfortunately, it is doubtful that many external reviewers would be willing to commit the large amounts of time needed to achieve the deep understanding of underlying models which is required to effectively validate a large-scale traditional simulation. We have become convinced that the time-intensive nature of validation is a major contributor to the credibility problems experienced by simulations in our field of Electronic Warfare.

Why does validation tend to be such a laborious process? We have identified three factors which we feel are especially significant:

(1) There is no framework for applying tests of validity. In other words, we probably need to have an explicit model of the modelling process itself. Remember that we learn about any system (including a simulation) by building models, by subjecting our models to analytical and empirical study and test, and by modifying our models based on the results of these tests. Consequently, validation tests are really only likely to be meaningful in the hands of an intelligent reviewer with expert knowledge of both the simulation and of the system being simulated. Most so-called tests of validity are very dependent on the context of the simulation, and hence difficult to generalize. We are currently lacking the kind of "software testbed" structure which would make these tests useful in a broader context.

(2) Validation often produces inconclusive or ambiguous results. The ambiguity may arise because the simulation was poorly designed, because the wrong kind of performance data was collected, or because we quite simply asked the wrong questions. Even if a simulation does "pass" all of the tests that one can conceive of (or afford!), there is still no guarantee that it is correct; one only knows that one has failed to prove it was incorrect. Moreover, if the simulation fails one or more of the validation tests, it may not be possible to find a cost-effective way to correct it.

(3) There are no software tools designed specifically to assist in the process of validation. The usual software development tools (e.g. editors, compilers, debuggers) are designed to manipulate programs; what we need are tools to represent and manipulate models. It is difficult to see how such tools could be "added on" to an existing simulation constructed along traditional lines. Consequently, we are led to conclude that simulations must be designed to be more open and articulate systems, capable of intelligent interaction with both designers and reviewers. This will require fundamental changes in the way that simulations are developed and maintained.

Currently, simulations are semi-autonomous entities which do not interact much with software development programs. We envisage a more integrated system of software development and support tools which will create a kind of "permanent environment" within which the simulation will be housed. Such environments would not only enhance the productivity of the simulation developer, it would also allow new users to more quickly achieve a level of understanding sufficient to engender confidence in the models. Moreover, such simulation support environments could also provide part of the validation framework we are lacking with current technology.

2.3 Requirements for an EW Simulation Environment

What features would we like to see in an EW programming environment? We will assume that any such environment should include a rich set of program development tools. We will consequently be more concerned with identifying software tools which would expedite the modelling process, provide a framework for simulation design, or assist in the validation of the finished product. These remarks will be drawn largely from our own experience. In most cases, they stem the hindsight that "if I had this tool, that job would have been easier". What follows is a compendium of such observations organized into related groups.

2.3.1 Simulations Should be Understandable and Verifiable

An EW programming environment should encourage a style of modelling which leads to a direct and meaningful correspondence between the subcomponents of a model and those of the real world system it represents. This requires a close and obvious relationship between the granularity of a simulation and the level of detail with which it models natural phenomena. Part of our motivation for wanting to build simulations within a programming environment is a desire to use the simulation itself as a vehicle for communicating with our peers, that is, as a way of storing and retrieving information about EW models. People tend to organize such information in an hierarchical fashion. It seems reasonable that EW simulations should take the same approach.

Indigenous support should be provided for the validation/verification process. For example, some form of automatic code generation is needed. Not only would it help to create more reliable code by eliminating the source of most programming errors (i.e. the programmer), it is also philosophically consistent with our goal of making the simulation designer more efficient. We envisage that the environment would ultimately include a library of "standard" components. Most of the analysis routines provided with current simulations are designed to analyze the behaviour of the system being simulated. We tend to use side-effects of these routines to extract information about model behaviour. EW programming environments should provide analysis routines designed specifically to examine the structure of the model itself, and to explain this structure to the user. One possibility would be to include an expert system which could deduce whether new models were "plausible" using a knowledge base containing "legal" ways to assemble the standard components in the library. This inference system could then be used by both designers and reviewers to find non-standard constructions.

The EW programming environment should include built-in event-logging machinery. Software to compile and analyze event histories seems to be a major component of most EW simulations, requiring considerable effort on the part of the simulation designer. Technology limitations in most current EW simulations mean that event-logging software must be "hard-coded" into procedures (e.g. as subroutine calls). As we learn more about the system being modelled, our perceptions change, with the result that simulation code is continually being modified to include new event-logging procedures. Finally, a point is reached when large new programs are required to post-process the enormous amounts of data being generated. We believe that more flexible monitoring capabilities should be provided, perhaps by including an "event editor" which allows the user to interactively define the event histories he wishes to record.

2.3.2 Simulations Should be Easy to Build and Use

Domain experts (like EW engineers) must have direct contact with the simulations which realize their models, without having to cope with the overhead associated with traditional simulation methods. After all, the end goal is not to create a simulation, but to gain an increased understanding of the systems being designed, built, and tested. As has been pointed out, this understanding is gained through model building. Simulations enter into the picture because they are often the only way to interact with complex models which do not have simple, closed form solutions. The EW analyst is now forced to choose between good programming techniques, which are time-consuming and which for a complex model require a team of programmers to implement, and the immediate feedback which comes from a simpler, "quick and dirty" approach with "throw away" code. Unfortunately, the simple approach is often inadequate, and the throw away code is hardly ever thrown away!

Consequently, an EW programming environment should be an open and articulate system: it should respond to user queries about what it knows, and be able to explain its own behaviour as much as possible. It should be able to represent and display the information it contains within a number of alternate views, each designed to interact with a particular class of user/modeller using a familiar symbology. For example, descriptions of EW receiver models might be accessible as either logic flow charts or as system block diagrams, both generated automatically from the same computer code. Programmers would be more comfortable with the flow charts or with the code itself, while systems engineers would likely prefer the system block diagrams. Notice that either representation is a "true" description of the underlying model. (This is analogous to expressing the same concept in different languages.) This capability to maintain a number of user-oriented views is especially important if one wishes to draw experts from other related areas into the modelling and validation process.

As much as possible, modellers should be freed from the necessity to explicitly manage overhead. A large portion of most current EW simulations seems to be caught up with managing overhead: the user interface, the organization of procedural knowledge and data, event-logging and statistical analysis of recorded events, etc. Standardizing user interfaces alone should result in considerable savings in time and energy. This can be accomplished by assigning the responsibility for managing application specific behaviour to the application module; in this type of architecture, the user interface is concerned only with system-level behaviour. Built-in software configuration control (sometimes referred to as "change management") is essential. Individual "tailoring" of the environment to support specific applications can then be encouraged, since the differences between the "standard" configuration and that supported locally will always be automatically documented. This approach permits us to be flexible in the face of new requirements while still retaining the benefits of known standards.

2.3.3 Summary

In summary, an EW programming environment should:

1. support a direct and meaningful correspondence between a simulation model and the real world;
2. provide explicit support for the validation/verification process;
3. provide built-in event logging and analysis without requiring explicit code modification;
4. allow easy access to the simulation for non-programmer domain experts;
5. be an open and articulate system, capable of supporting multiple viewpoints;
6. relieve the user of responsibility for explictly managing overhead by supporting standard interfaces and automatic software version control.

While unrealized as yet, we believe that such a system is well within the current state-of-the-art in programming environments. It has seemed to us for some time that the nub of the credibility problem is that one is forced to grasp the mechanics of a simulation before one can comprehend the underlying models. This is unfortunate, because EW specialists already have a basic comprehension of the fundamentals of their discipline; understanding how the simulation works is almost always the most time-consuming part of the validation exercise. Consequently, we were led to ask whether it might be possible to design a simulation which enforced a separation between the EW model and the purely mechanical aspects of the simulation, without imposing a rigid structure on the designer. The concept of creating an object-oriented EW programming environment, as discussed in the next section, grew from this requirement. We believe that the development and use of such environments will lead in the long term to better simulations, improved validation, and, ultimately, to real credibility for EW simulations.

3.0 OBJECT-ORIENTED EW MODELLING

Although there are undoubtedly a number of ways to implement an EW simulation environment which addresses the requirements laid out above, we advocate an object-oriented approach. An object is a software construct which specifies in a single logical unit both data and all of the procedures which can manipulate that data. An object can only be accessed by sending it a message (executing a procedure); the external view of an object is its protocol, i.e. the set of messages to which it can respond. This encapsulation is typical of object-oriented systems, and is the underlying reason for the modularity and the robust response to change which object-oriented systems provide. Also intrinsic to most object-oriented systems is the notion of class. A class is, in effect, a template for a particular type of object. Classes are organized hierarchically in a tree-like database. Data structures and message protocols are inherited from higher level classes unless they are explicitly changed.

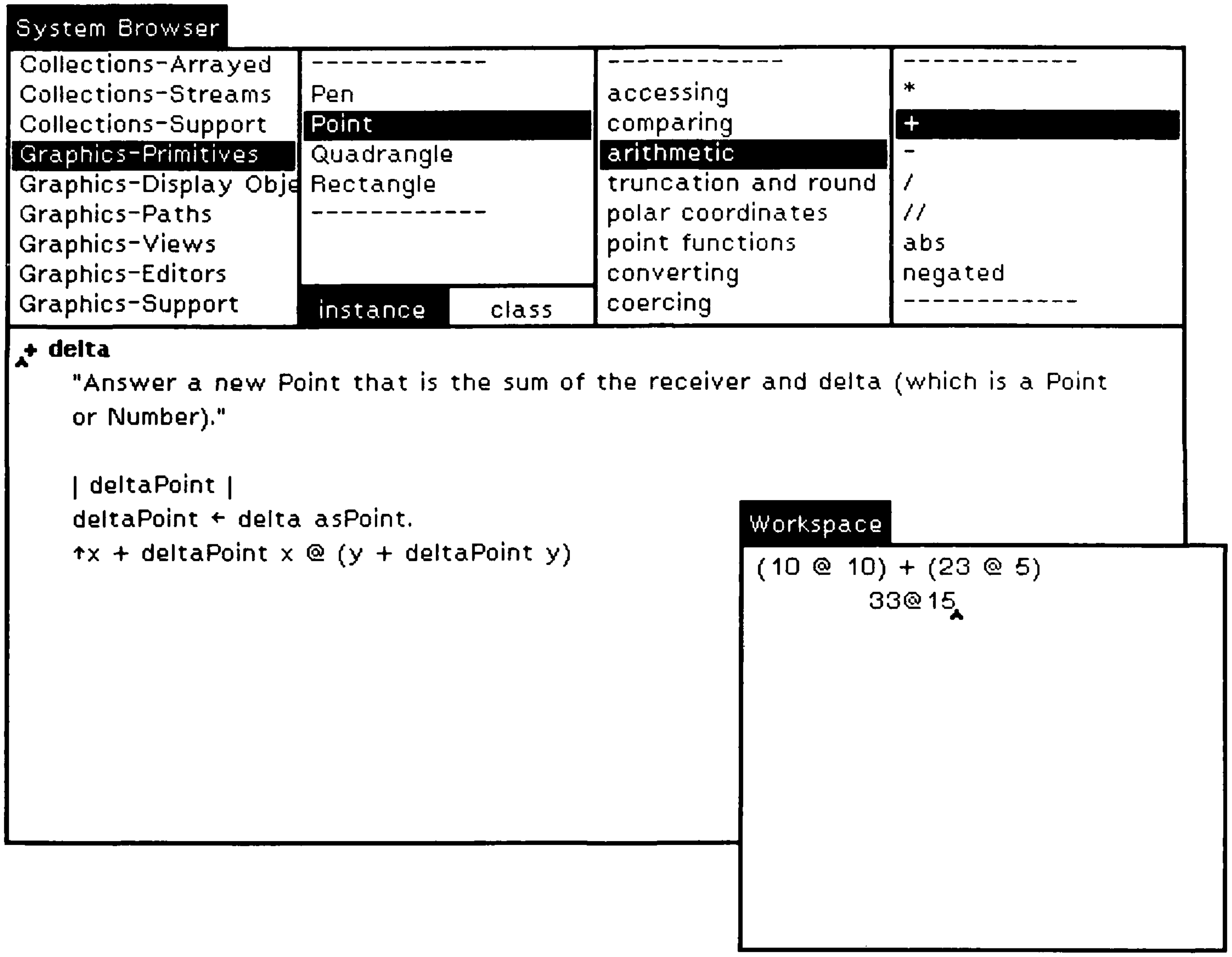
Although the SIMULA programming language was the precursor of object-oriented programming, it was the Smalltalk project at the Xerox Palo Alto Research Center (PARC) which, directly or indirectly, brought most of these concepts to maturity. The expressive power of the object/message-passing paradigm, as demonstrated by the Smalltalk-80 implementation [2], has been both persuasive and pervasive. Objects are now supported in one form or another in a number of programming languages and applications. However, Smalltalk is still the only programming language to use the object as its basic building block. In every other case, objects are built from other, more primitive, data types. Consequently, we believe that Smalltalk is the best base on which to build an object-oriented application. With this in mind, we are developing an extension of Smalltalk (called Actra) designed to run concurrently on the multiprocessor testbed described in section 2.0 [1].

The object-oriented approach provides a framework which supports a direct and meaningful correspondence between our models and the real systems we are modelling. Consequently, we have an expressive, natural way to represent the inherent structure in the EW problem. By using classes and inheritance we can deal more effectively with the complexity of EW knowledge by representing it in a layered fashion. It will be recalled that one of our objectives was to eliminate the need for routine programming. Since object-oriented "programming" tends to consist of modifying existing classes to create new subclasses, it represents an important interim step towards this goal. In fact, when simulating with objects, most of the effort is devoted to identifying and defining the new classes required to realize the abstract model.

A number of earlier EW simulation systems have either directly or indirectly influenced our design. Many of our ideas were formed as a consequence of working for several years with the U.S. Naval Research Laboratory's STEWS (Simulation of Total Electronic Warfare Suites) package. We have also had the benefit of a number of other examples of object-oriented simulations to draw on, especially the system described in [6] by Cunningham et. al., and ROSS [3] and SWIRL [4], which model tactical air engagements and have a significant EW component.

3.1 The Smalltalk Programming Language

Since the Smalltalk language is not yet widely used, we will include a brief introduction to the language and its accompanying programming environment. A complete description of Smalltalk is given in [2]. Smalltalk is usually run on a high-performance workstation equipped for high resolution bit-mapped graphics and a mouse input device. Personal computer implementations have only recently been available. The Smalltalk programming environment supports a menu-driven, multiple window format reminiscent of the Apple Macintosh (which is not surprising, since the Macintosh interface borrowed philosophically from the Smalltalk user interface). A typical Smalltalk display screen is shown in Figure 4.



```
+ delta
    "Answer a new Point that is the sum of the receiver and delta (which is a Point
    or Number)."

    | deltaPoint |
    deltaPoint ← delta asPoint.
    ↑x + deltaPoint x @ (y + deltaPoint y)
```

```
(10 @ 10) + (23 @ 5)
        33@15
```

FIGURE 4: Typical Smalltalk Display

Smalltalk has a completely integrated programming environment; there is no distinction maintained between the "language" and an "operating system". The Smalltalk system essentially consists of a large collection of objects. Viewed heuristically, each object has two main parts: its data structure (i.e. how it is represented and stored in the computer), and its "method dictionary". The method dictionary is a list of all the procedures (or "methods") the object knows about; these methods are indexed by a "selector". All computation in Smalltalk takes place as the result of message-passing between objects. A message usually consists of a receiver (i.e. the object to which the message is addressed), a selector (the index into the receiver's method dictionary), and, optionally, arguments to the selector.

To illustrate these ideas, let's consider a simple example. A Point object is stored as an array of two integers; points in Smalltalk are written as

x-coordinate@y-coordinate

e.g., 10@10, 23@5. One of the procedures associated with the class Point is point addition; its selector is "+", and it expects one argument, another Point. To add two points, we send the message "10@10 + 23@5". The receiver, 10@10, looks up "+" in its method dictionary, and finds a procedure which, in effect, says "take the argument (23@5) of this message and add it to yourself".

Smalltalk objects have associated with them the notion of class. A class is a set of objects with similar attributes and behaviours. Any specific member of a class is referred to as an "instance" of that class. In our example, Point is a class, while 10@10 and 23@5 are instances of the class Point. The class is the reference for data structures and method dictionaries; the instance stores the actual values which define a member of the class. Smalltalk supports inheritance; that is, classes can inherit data structures and procedures from parent classes, usually referred to as "superclasses". The child class, or "subclass", can add new data structures. It can also add new procedures or modify those of its superclass. Inheritance is a powerful mechanism for sharing code; without it the overhead associated with object-oriented programming would be prohibitive.

The Smalltalk programming environment, then, is simply a large collection of useful classes: Numbers, Points, Views ("windows"), Menus, and so forth. One of the most useful features provided by the environment is the System Browser (which is, of course, just another object). The System Browser is the central window shown in Figure 4; it is divided into four small "list views" in its upper half, together with a "code view" in the lower half. The System Browser is a kind of database management system which provides access to all the objects in the environment. Smalltalk classes are organized into related groups called "categories". These categories are shown in the Browser's upper left list view. By moving the mouse, one can scroll through the categories and select one. In Figure 4, the "Graphics-Primitives" category has been selected. This is indicated by dark high-lighting around the selection.

The list view to the right of the category view shows all of the classes in the selected category. In Figure 4, we see that one of the Graphics-Primitives classes is Point, which has also been selected. Just as classes are grouped into categories, so too are the methods associated with a particular class. In the third list view (going from left to right) we see the message categories, or protocols, associated with the class Point. The "arithmetic" protocol has been selected, and a list of the selectors in this group appears in the right-most list view. We see that all of the usual arithmetic operations which make sense when applied to points are present. In particular, the "+" operation has been selected, and the code which defines the "+" method is shown in the code view (i.e. the bottom half of the Browser).

The Browser can be used not only to view existing objects like Point, but to create new classes and messages as well. The code view portion of the Browser is a complete text editor. New code written with the Browser can be automatically added to the environment by selecting the appropriate menu item. Smalltalk menus are "pop-up" menus, i.e., they appear on the display screen at the current cursor position. We will see more examples of the use of menus to build applications later. For now the key points to keep in mind are:

(1) objects consist of data and procedure combined;
(2) objects are organized into hierarchical classes;
(3) using the Browser we can easily create new classes and messages.

3.2 An Overview of the Simulation Architecture

Smalltalk programming consists essentially of creating new classes and modifying existing ones. Consequently, in order to implement an EW simulation environment in Smalltalk, it was necessary to identify and define the collection of new objects which together comprised the simulation system. As illustrated in Figure 5, these new classes divide naturally into two broad categories: those used to implement various aspects of an EW simulation, and those used to construct the user interface.

3.2.1 The Environment Model

Most EW simulations are concerned with modelling the interactions between electronic systems and the so-called "electromagnetic environment" generated by such RF emitters as radars, jammers, and so forth. The simulation may be designed to investigate large-scale effects (e.g. how an ESM receiver responds to a large number of radars), small-scale effects (e.g. interactions between a missile seeker radar and a defensive jamming system), or both. Our current interests are restricted to many-on-one ESM receiver simulations (i.e. large-scale), but the methodology discussed here is equally applicable to the small-scale case.

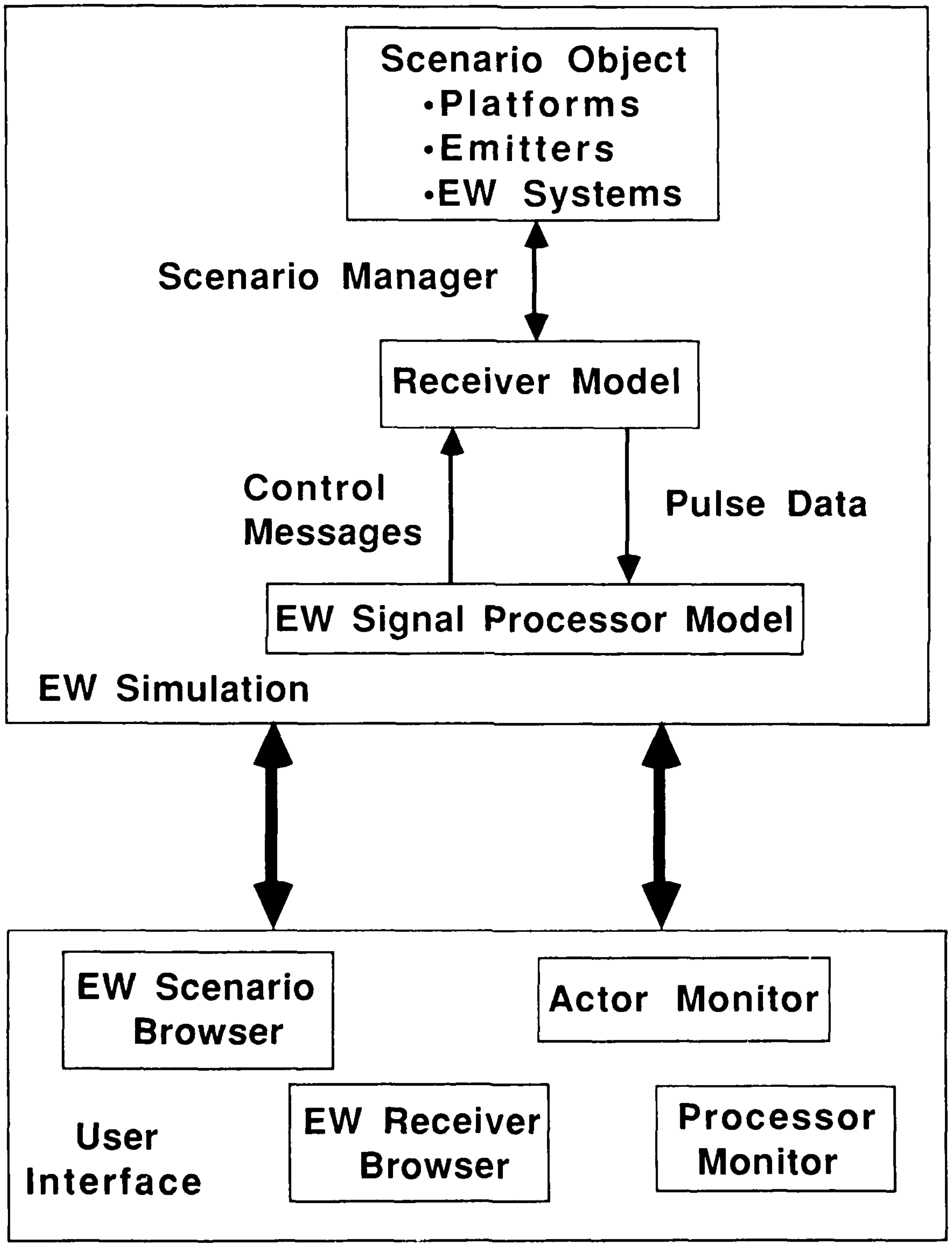


FIGURE 5: Structure of Object-Oriented Modelling System

The electromagnetic environment is described by a Scenario object, i.e., an instance of class Scenario. As shown in Figure 6, Scenarios are organized around Maps. Each Map describes an area within which the objects in a particular Scenario move. A Map stores its origin, scale, and units, as well as a name and comment. In addition, each Map has a form containing a bitmap used to display the Map on the screen. Each Map contains a list of Scenarios defined on that Map. A Scenario has associated with it a name, a time interval over which it takes place, and a comment. Most objects in the simulation environment have a comment "slot" which the simulation designer can use to document his design decisions. For example, a Scenario comment may include a written overview of the scenario, describing the general situation and the objectives of the opposing commanders.

A Scenario includes a list of Platform objects. As shown in Figure 6, each Platform has a name, an indication of type (e.g. Surface, Air, etc.), and a disposition (e.g. Friendly, Hostile, Neutral). In order to be able to display the platform's location on the Scenario's map, it has an icon (a small graphical symbol which can be used to represent the platform). Each Platform also has a trajectory, which is essentially a list of times and positions which define the Platform's motion during a simulation. Finally, the Platform has a list of Emitter objects. As indicated in Figure 6, each Emitter object has a number of state variables which define its dynamic behaviour. An observer located anywhere on the scenario's map can ask an Emitter to use this information to compute the signal power which the observer will receive from the Emitter at a particular point in time.

A instance of a Scenario can be thought of as a database containing information about the electromagnetic environment. Heuristically one tends to think of emitters as being a collection of (nearly) independent objects acting in parallel, suggesting that emitters should be modelled by a number of asynchronous processes Unfortunately, this generates more computational overhead than can be easily managed with our current implementation. Consequently, we have created an abstract object called a Scenario Manager, whose function is to provide an interface between the emitters in a Scenario and any EW equipment models in the simulation (see Figure 5).

We will usually refer to objects such as the Scenario Manager as "actors". An actor may be thought of as a process which has a list of tasks to perform (its "script"); alternatively, one may think of an actor as a dynamic object which is encapsulated within its own process. The other principal actors in the current implementation of the simulation environment are Receivers, Data Buses, and Processors. In general, actors can be decomposed into a number of "lower level" objects. These lower level objects may communicate directly with each other; however, to send messages to objects outside the actor's domain of definition, they must communicate via the actor. This discipline leads to "modularity in the large" which complements the object-oriented "modularity in the small" provided by Smalltalk.

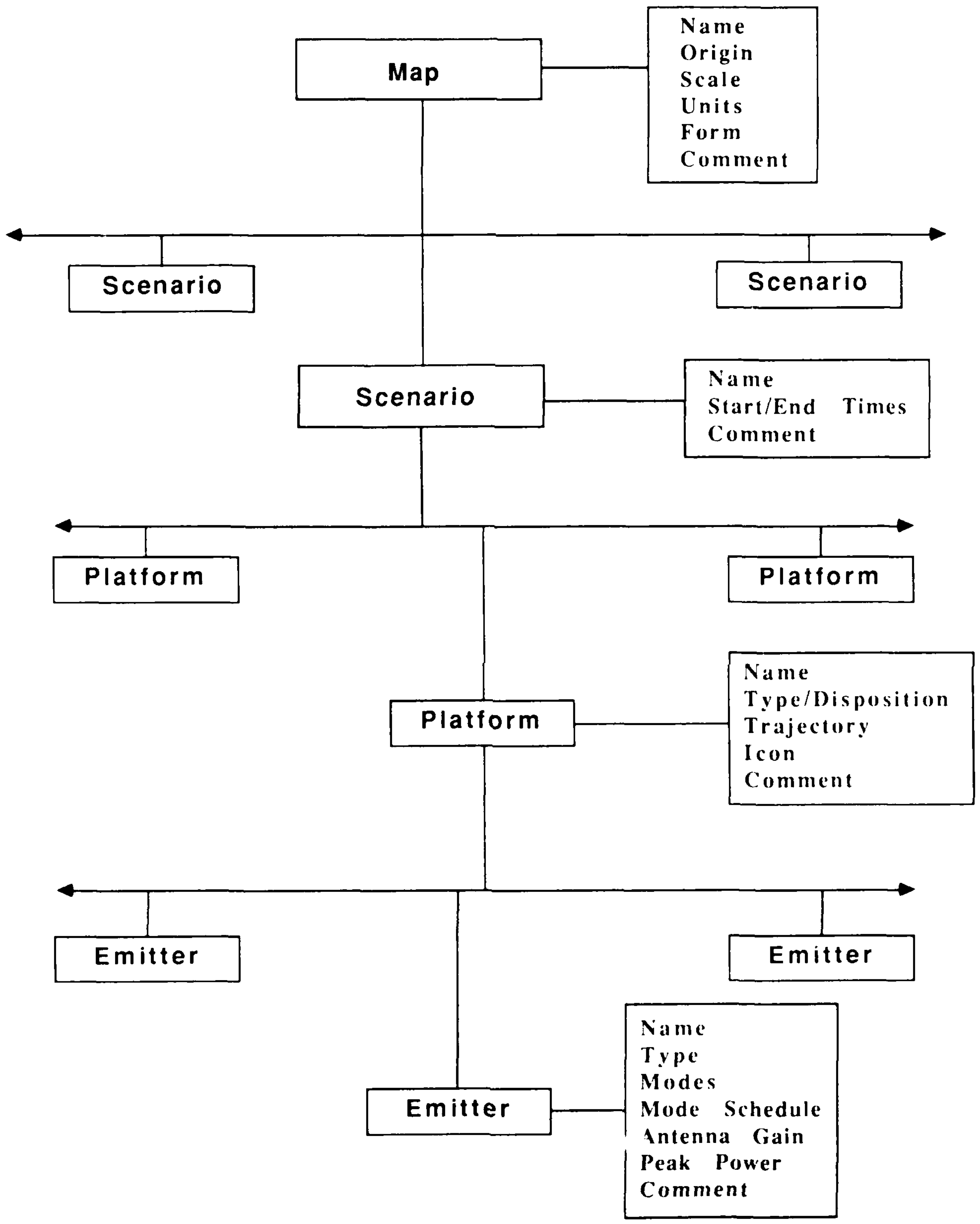


FIGURE 6: Scenario Database Structure

3.2.2 The ESM System Model

A detailed description of the Receiver and Processor actors is shown in Figure 7. Receiver objects live on one of the platforms in a Scenario (the Associated Platform); they receive pulse data from a Scenario Manager. Signal processing within a Receiver is modelled by a Signal Flow Net object: when the Receiver receives a Pulse object from a Scenario Manager, it sends a " process this pulse" message to the Signal Flow Net. A Pulse object is essentially a list of parameter values, specifying the RF (radio frequency), TOA (time of arrival), PW (pulse width), DOA (direction of arrival), PA (pulse amplitude), and source emitter. The Signal Flow Net is a network of abstract devices: Filters, Detectors, Transformers, and Bus Terminators. By interconnecting a number of these abstract devices we can quickly build functional models of most receivers. Although the current implementation does not model stochastic effects, these could be added with relatively minor modifications to the overall design.

A Filter interrogates the Pulse to see if a particular parameter value is within an acceptance gate, and, if so, it passes the Pulse on to the next device in the net. Since Filters can be defined on any parameter, we can model not only RF filters but also such components as directional antennas and threshold detectors using Filters. A Detector measures a parameter value and performs some action (e.g. record the value, feedback to another device, etc). A Transformer performs a linear transformation on a particular parameter value, allowing us to model such components as mixers and amplifiers. Bus Terminators provide a standard interface to a Data Bus object. As indicated in Figure 7, our system can simulate a computer which employs a number of processors running asynchronously. Our computational model is based loosely on the Harmony operating system. Harmony is a realtime operating systems kernel developed by Gentleman [7][8]; it belongs to a family of widely respected message-passing operating systems which are beginning to see broad use in the industrial and academic communities.

Harmony is a shared-memory multiprocessor real-time kernel. Each processor contains a copy of the Harmony kernel and executes a set of tasks which reside in its address space. Each processor is capable of interrupting any other processor. Any task can send a message to any other task in the same or a different processor. The location of the task does not change the application program. Since both local and remote tasks are referenced in the same way, tasks can be moved to different or more processors without modifying the application software. The tasks which are executed in a given processor are determined at configuration time. Any number of instances of a task can be created and destroyed at run time. Tasks execute with a fixed stack space, but may request and release working memory in the processor where they execute.

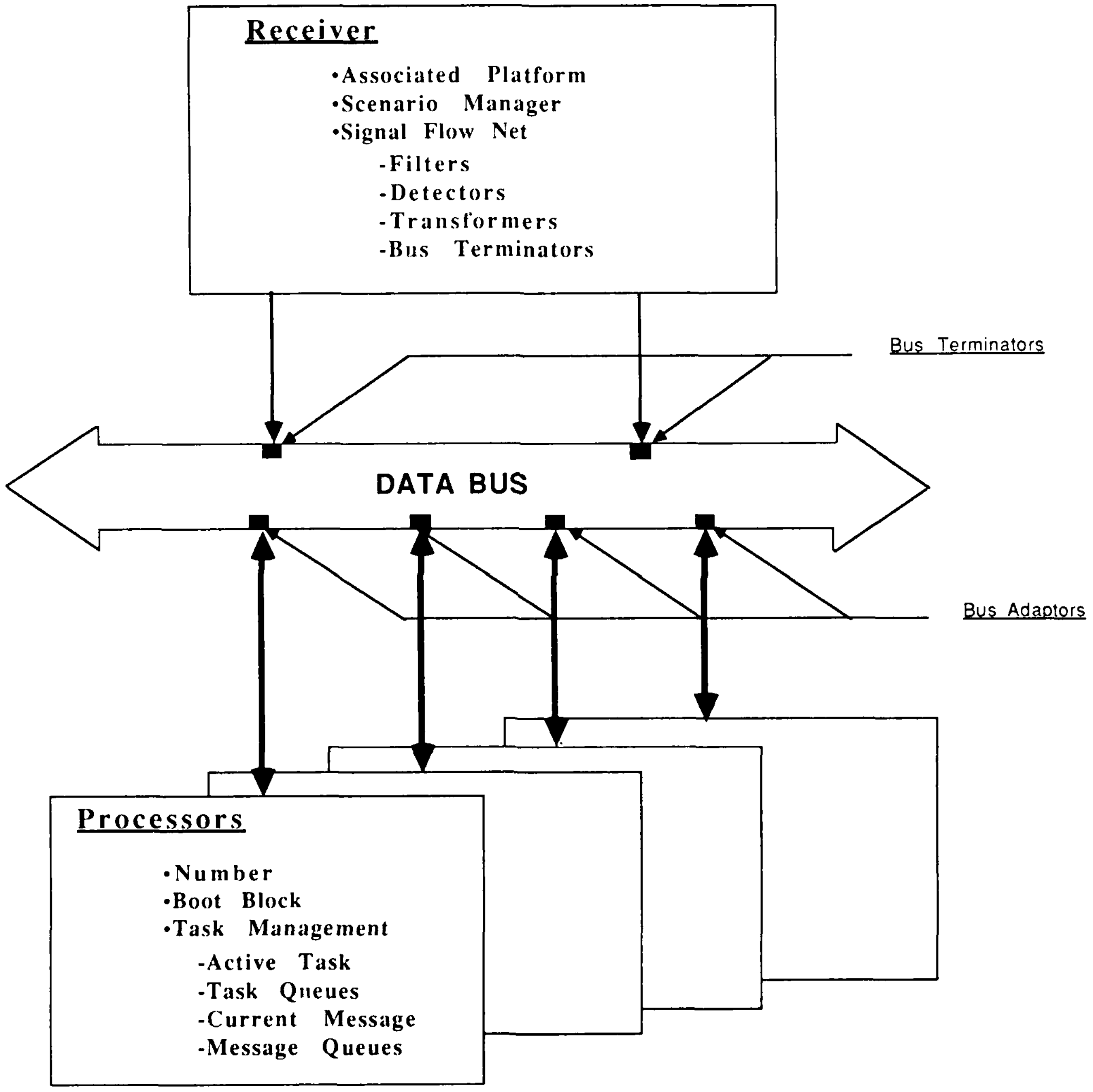


FIGURE 7: Receiver/Processor Structure

The bulk of intertask communication is accomplished with three message passing primitives: send, receive, and reply. The "send" message causes the executing task to block or suspend until it receives a reply, whereas "receive" causes the executing task to become blocked until a corresponding request has been received. This simple and straightforward protocol is augmented by two special forms which implement a non-blocking receive and interrupts. Additional primitives for creating and terminating tasks and supporting stream-oriented input/output are also provided.

The main objects used to simulate multiprocessor computers in our simulation package are Data Buses, Processors, Tasks, and Messages. A Data Bus object has a collection of Bus Terminators and Bus Adaptors. Its function is, quite simply, to model the movement of data on the bus according to some bus arbitration protocol. Each Processor object has a unique identifying number,a boot block which contains initialization information, priority queues for tasks and messages, and pointers to the currently executing task and the last received message. Message objects have state which records the sender and receiver of the message, the times sent and received, a message type symbol, and the body of the message (which can, in principal, be any object). Task objects are independent processes which communicate and synchronize with message-passing based on the Harmony model. We model a particular task in a Harmony-like system by creating subclasses of Task which include the appropriate protocol for the task being modelled.

3.2.3 Simulation Time Management

The execution of a particular simulation is driven entirely by message passing between the actor objects; message passing between actors follows the same Harmony-like protocols used for Tasks. The processes associated with the various actor objects are first initialized. Each actor object has a local clock; in addition, there is a global clock kept for the system as a whole. Time management is an important issue in an actor based simulation. When an actor receives a message from another actor, the local clocks of the sender and the receiver are synchronized. However, we have to guard against creating paradoxical situations.

For example, consider three actor objects A, B, and C, and suppose local Time is a message which returns an actor's current clock value. Suppose further that

A local Time $<$ B local Time $<$ C local Time

i.e., actor A's local time is in the past relative to actor B's local clock, and actor B's time is in the past relative to actor C's. Since the simulation is running asynchronously, it can happen that A receives a message from C, updates its clock to match C's, and proceeds to act on the message. Meanwhile, B sends a message to A, which A is not available to process because it is busy working on a "future" message.

Our solution to this problem is to maintain a global clock which is defined to be the minimum time on the clocks of all actors currently blocked on a "send" message. Message transactions are not initiated until the sender's local clock matches the global clock, and are not completed until the receiver's time equals or exceeds global time. This guarantees that an actor blocked in "receive" loops will not unblock until there are no potential senders in its "past", only receivers. Moreover, these receivers can only become senders when they are unblocked, at which time their clocks will be updated past the current global time. Notice that only actors currently engaged in a message transaction require synchronization. Otherwise, any side effects resulting from the actor's processing are internal to itself, and cannot affect any external objects in the rest of the system. Actors blocked in "receive" loops will be unblocked as soon as a sender matches the global clock.

We will illustrate how a typical simulation might proceed by considering an example based on our ESM multiprocessor simulation. In the current implementation, the simulation is initiated when tasks associated with data acquisition send messages to the Receiver object requesting pulse data over some time interval. The Receiver asks the Scenario Manager for any pulses which would have been generated by emitters in the scenario during this interval. Each of these pulses are processed by the Receiver, and any which are accepted, i.e. would have been "intercepted and detected", are sent to the requesting tasks via the Data Bus object. This is actually a somewhat simplified overview of what actually takes place, since special hardware features planned for new systems are also simulated. However, it serves to provide a general idea as to how the objects in the simulation interact. In the next section, we will discuss the user interface facilities provided to support direct interaction between the simulation objects and the designer/user.

3.3 An Overview of the User Interface

Many of the desirable features for an EW simulation environment which were discussed in section 2.2 focused on the interface between the designer/user and the simulation. It was suggested that the simulation should be open and articulate, that it should employ standard interfaces and version management, that event-logging and analysis should not require explicit code modification, and that non-programmers should have easy access to the simulation. Our initial approach to addressing some of these requirements is to try to explicitly separate some aspects of the user interface from the main body of the simulation, as indicated in Figure 5.

We envision the user interface as a collection of software tools which are designed to interact with specific classes of simulation objects. These tools are themselves objects, and hence can be easily modified (by creating new subclasses) to support new requirements. Each of these objects have been designed as independent elements following the object-oriented philosophy. Hence the tool set can be expanded, and existing tools can be modified, with little danger that unexpected interactions would render existing software unusable.

We are using two types of interface objects at the moment, Browsers and Monitors. Both provide access to the state of simulation objects: browsers support extensive off-line (i.e. while the simulation is not running) interaction between the browsed object and the user, while monitors are used during the running simulation to examine dynamically changing states of simulation objects. Two browsers are currently planned: a Scenario Browser which will provide an electromagnetic scenario generation and analysis capability, and a Receiver Browser, which will provide a capability to create EW receiver models directly from system block diagrams. A prototype of the Scenario Browser has been implemented, and will be described in some detail later in this section.

3.3.1 Monitors

Two different types of monitors have also been created. The Actor Monitor is a kind of "generic" interface designed to inspect the state of any actor. Processor Monitors are a subclass designed specifically for inspecting Processor objects. Each monitor creates an independent multi-paned window to display information to the user; the layouts of the windows associated with each monitor are illustrated in Figure 8. Actor Monitors show the name of the actor, the time on the actor's local clock (this pane can also show the global time), and the message state (i.e. Send, Receive, Reply). The "Catch/Throw" pane is a switch: in "Catch" mode all messages to and from the actor are intercepted and displayed in the "Message Inspector" pane. Outgoing messages can be edited by the user before being forwarded. In "Throw" mode message interception is disabled. The "Message Log" pane is a log of the most recent message traffic. It can be reviewed periodically by the user and, if desired, written to a file.

As shown in Figure 8, Processor Monitors have additional display panes which provide information about the internal state of the processor. Specifically, the "Active Task" pane indicates either the name of the task currently being "executed" or "Inactive" if no task is executing. The "Priority Queue" pane is a list of tasks ordered by priority which are waiting for the processor. The "Suspended Tasks" pane is a list of tasks which are blocked because they are waiting to complete a message transaction. The "Task Message Queue" is a list of messages which have been received by the Processor but not delivered (usually because the Task to which the message is addressed has not yet asked to receive it). Any task or message can itself be inspected by pointing to its name with the mouse and selecting the "inspect" option from a menu.

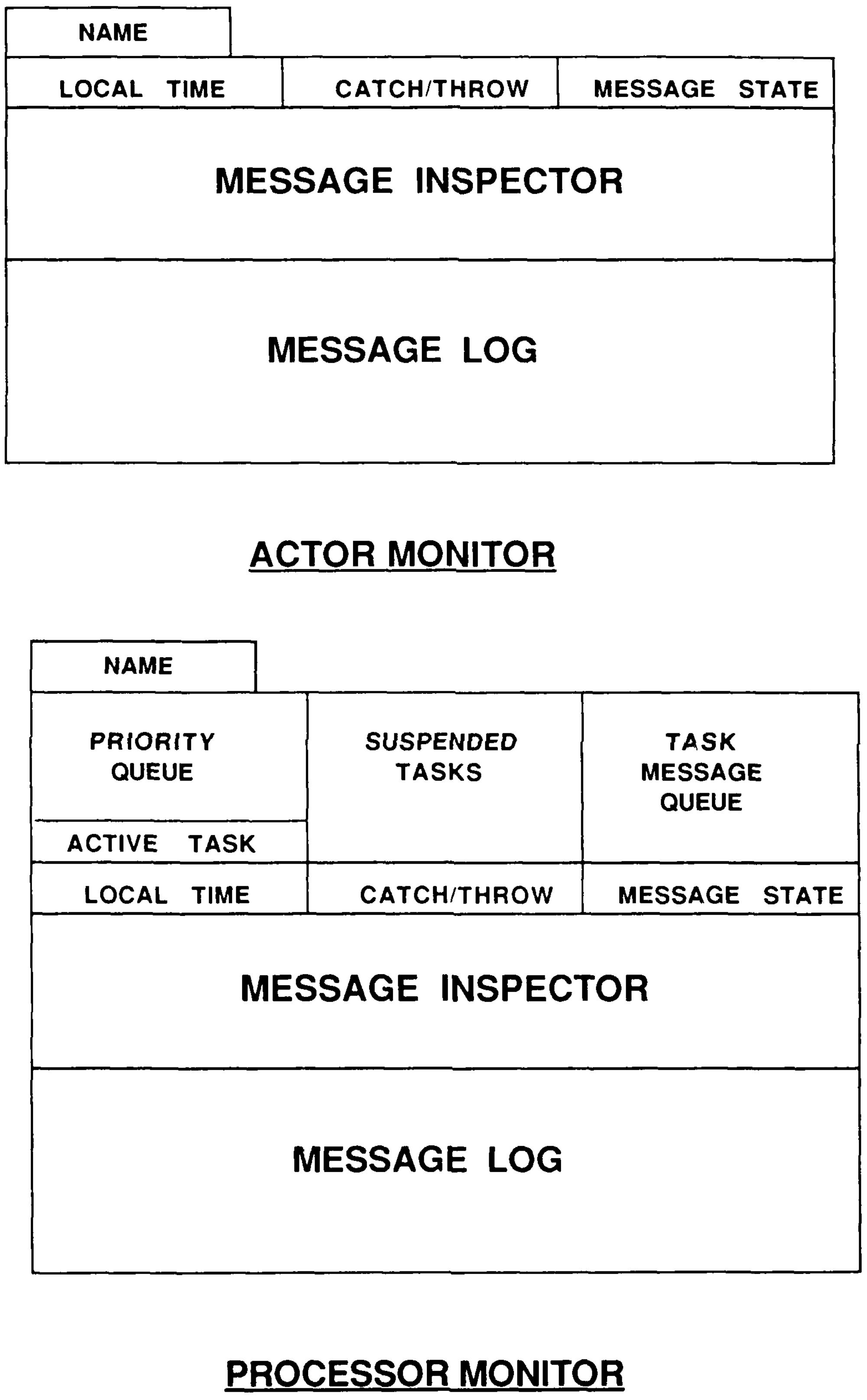


FIGURE 8: Layout of Monitor Displays

3.3.2 Browsers

The function of the Browsers is best described by following through an illustrated example which shows how a user would create an electromagnetic scenario using the Scenario Browser. Figure 9 shows the layout of the Scenario Browser. The Browser is organized around maps (upper left pane). Associated with each map is a set of scenarios (shown to the left of the map pane). Initially, the Browser indicates that the "Maritimes" map has been selected, and that it has one scenario called "Trials". The map itself is displayed in the right half of the Browser. The map pane supports both scroll and zoom operations, with the length and position of the scroll bars indicating the portion of the map being displayed. When the mouse indicator is moved into the map pane, it is displayed as a crosshair. As the crosshair is moved around on the map, the current position coordinates are displayed in the small pane at the lower left. In the figure, the coordinates "261 nmi @ 214 nmi" are indicated.

In Figure 10, the "Trials" scenario has been selected. The platform pane (directly below the map pane) shows a list of the platforms which appear in the scenario. Associated with each platform is an icon which shows the platform's position on the map. A platform may be selected by clicking the mouse button on either the platform name or the icon. In either case, both the name and the icon are highlighted, as shown in the figure (the platform Vancouver has been selected). When a platform is selected, any radars or EW systems on that platform are shown in the equipment pane (right of the platform pane). For example, in Figure 10, the only equipment on the platform Vancouver is the CANEWS ESM system.

Associated with each pane in the Browser are one or more menus which provide processing options for that pane. Menu items are selected by depressing the middle button, moving the highlight to the desired item, and releasing the button. Figure 11 illustrates a platform menu. The "add platform" option has been selected. Other options allow the user to set default units for speed, altitude, and depth, read a platform from a file, and "inspect" existing platforms. The purpose of the inspect selection will be described shortly with a specific example.

The Scenario Browser maintains a database of known platform and emitter types. Selecting "add platform" initiates a series of prompts aimed at establishing the name and type of the new platform, as shown in Figures 12-15. Using the platform type information stored in the database, a new instance of that platform is created and added to the scenario (Figure 16). The user is prompted for the initial position of the platform, and the basic platform parameters are displayed in a pop-up view. We next specify the course and speed for the new platform. As shown in Figure 17, menu options are provided to set the speed and altitude of the new platform, and to set a new position. Selecting the "set speed" option produces a "limited choice" menu, as shown in Figure 18. The range of possible speeds for this platform type is cxtracted from the database, and a sliding scale indicator is used to select a value in this range (fictitious values have been used in all examples).

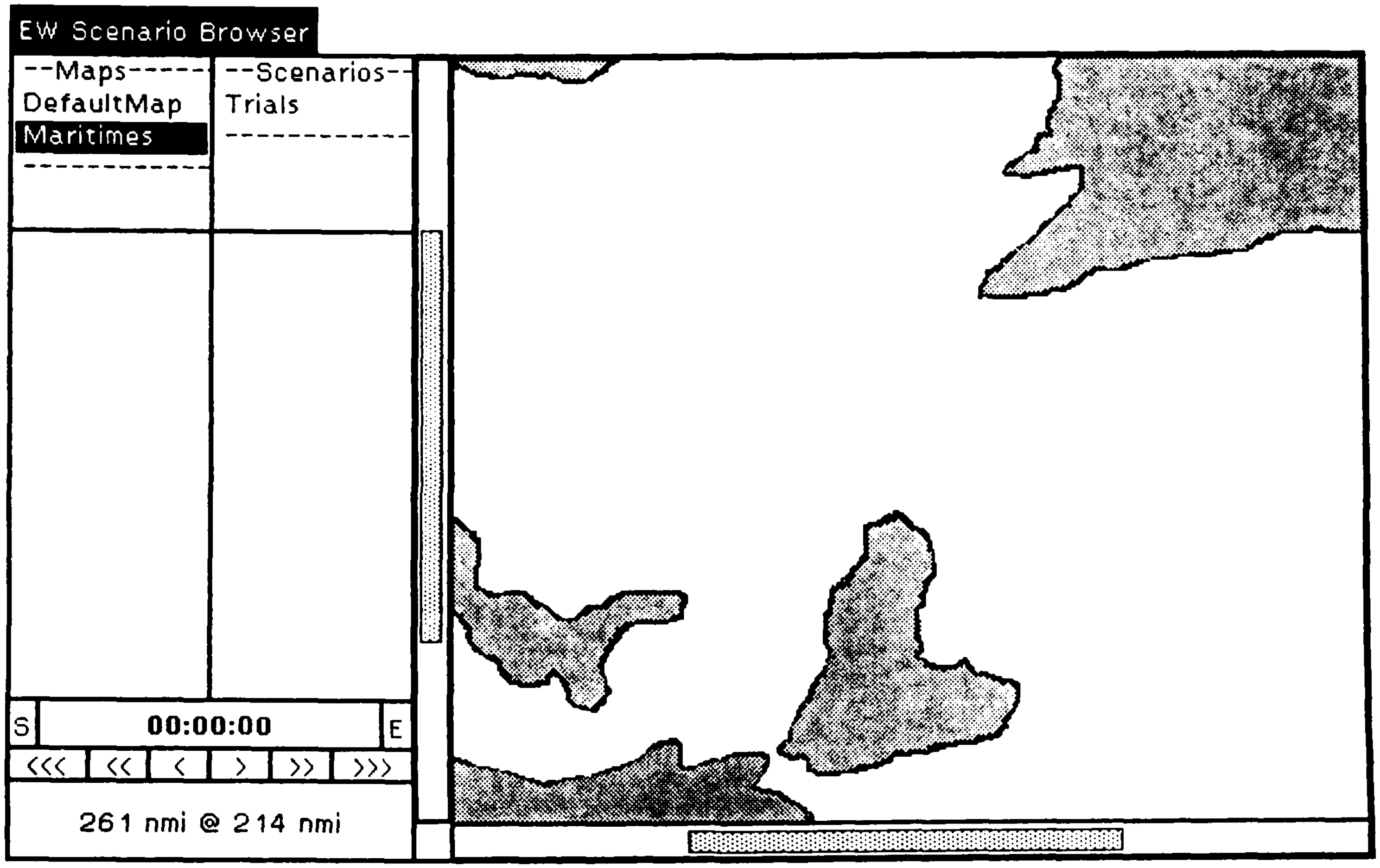


FIGURE 9: Scenario Browser Layout

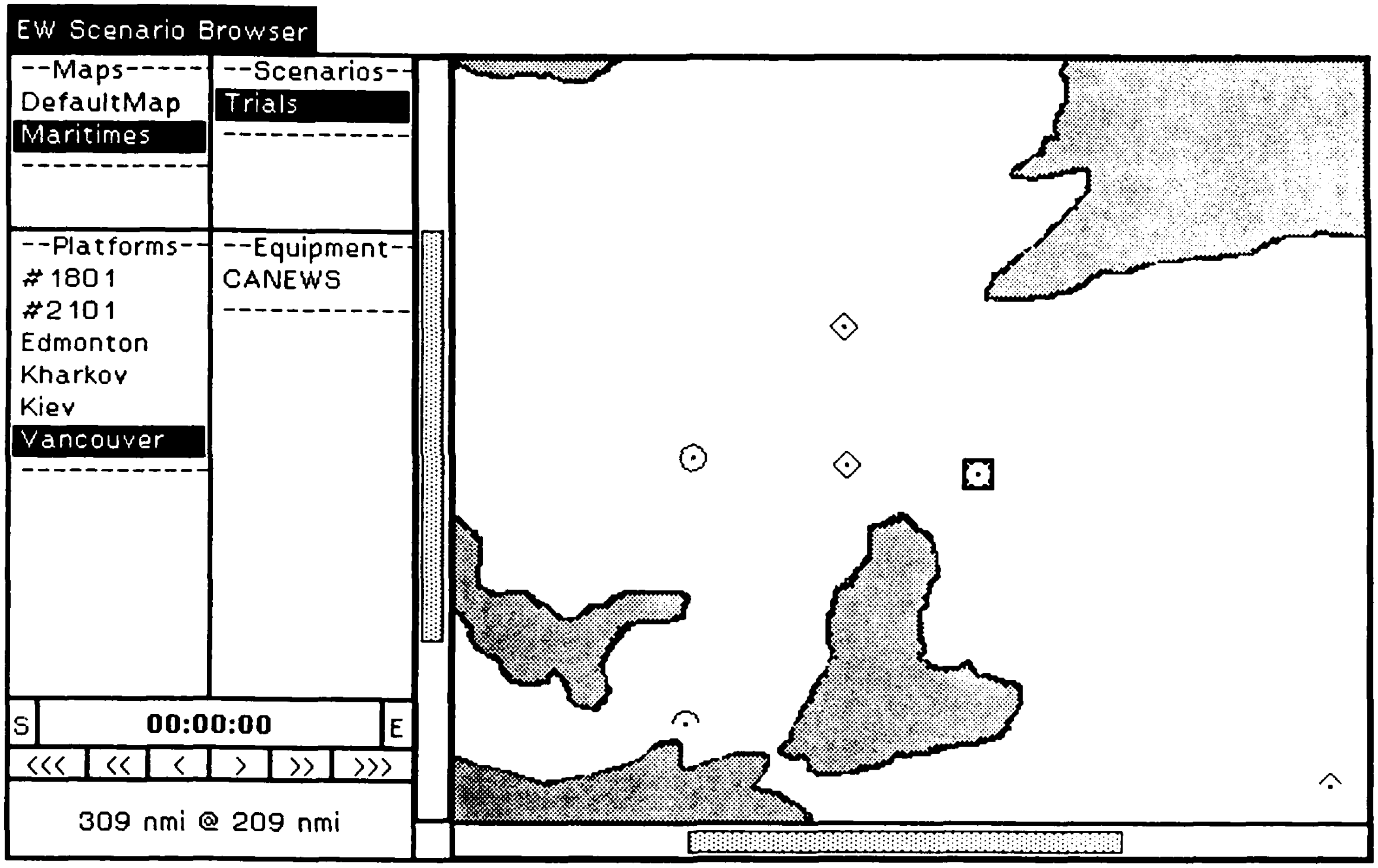


FIGURE 10: Scenario Browser: Selecting a Scenario

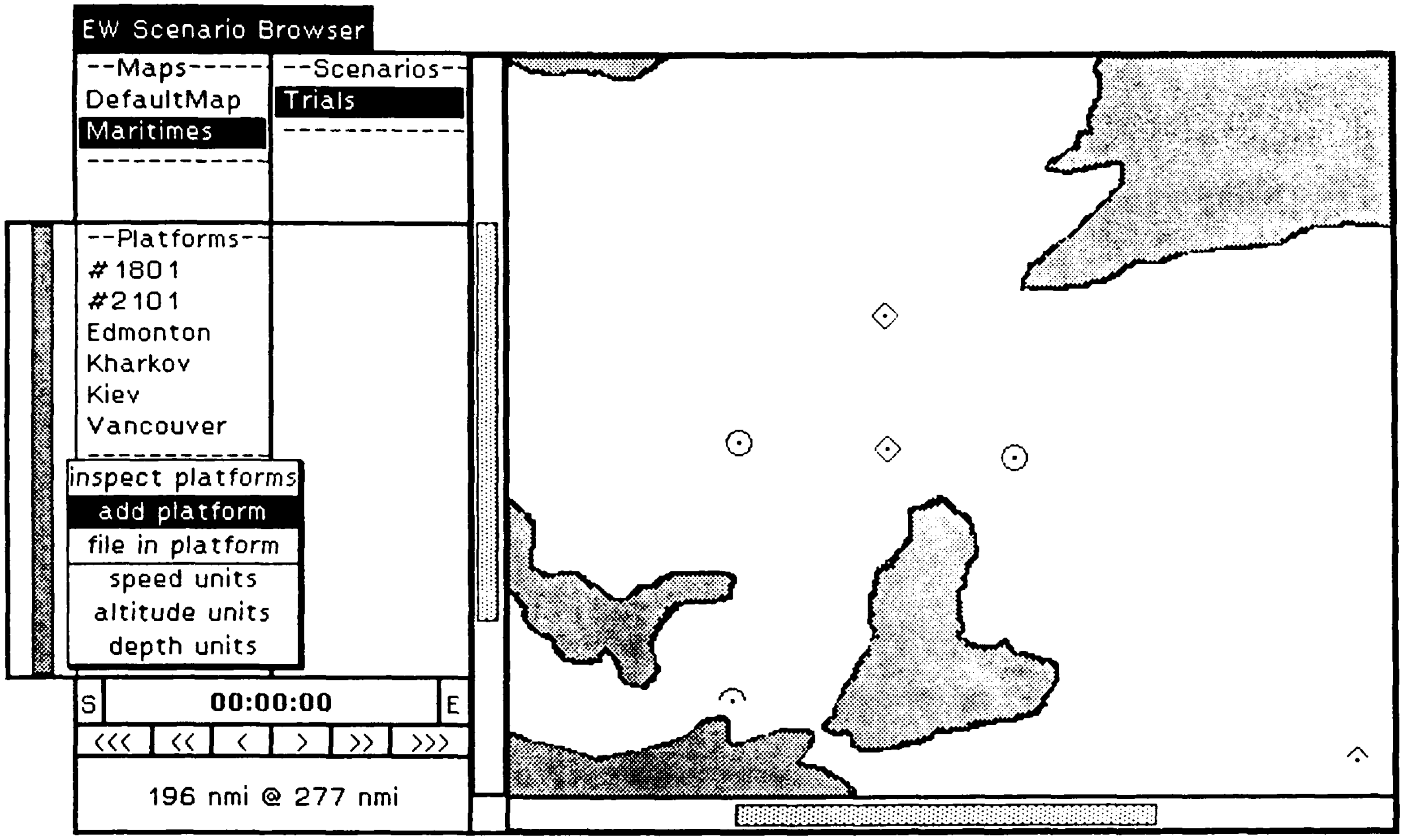


FIGURE 11: Scenario Browser: Adding a New Platform (i)

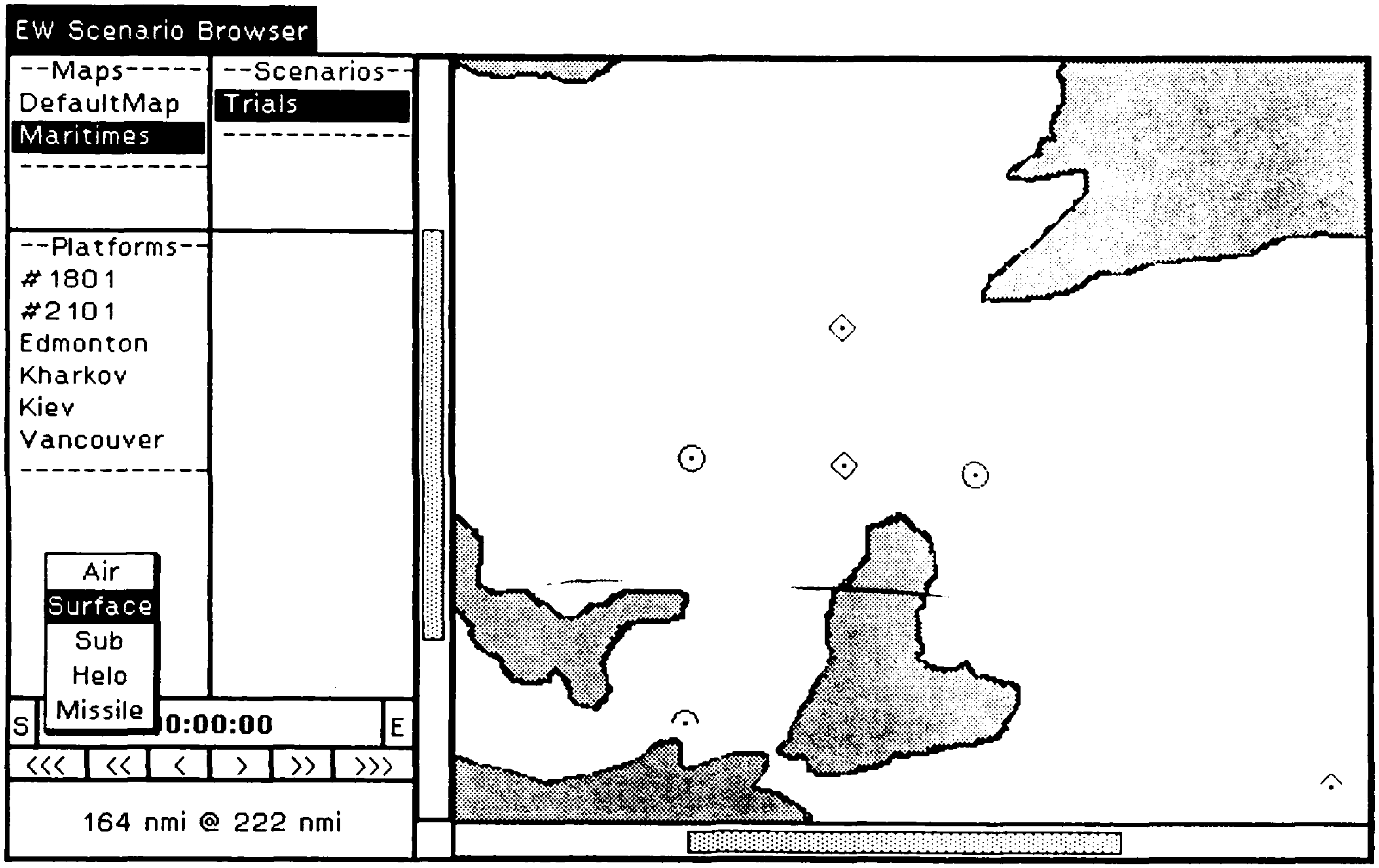


FIGURE 12: Scenario Browser: Adding a New Platform (ii)

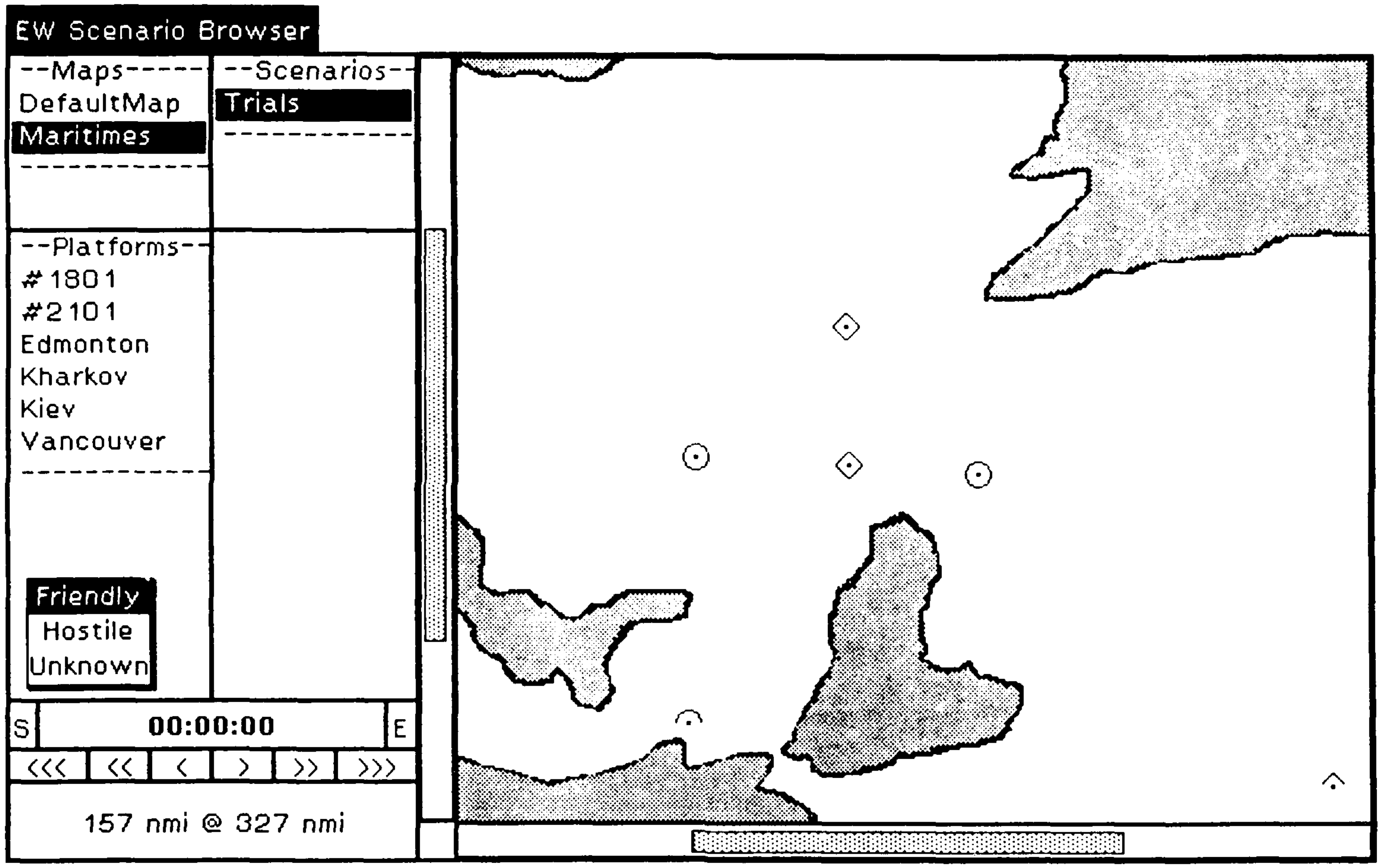


FIGURE 13: Scenario Browser: Adding a New Platform (iii)

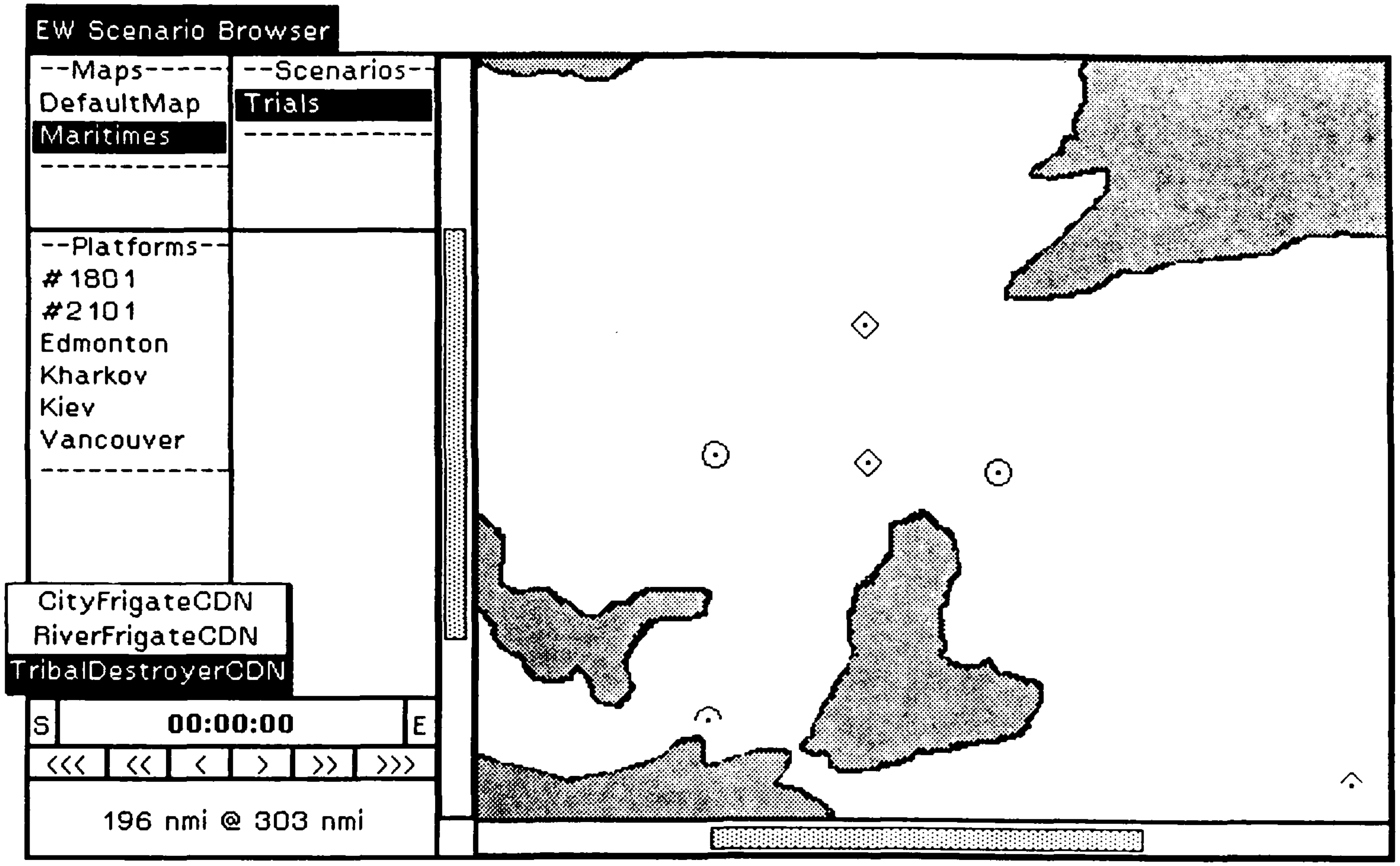


FIGURE 14: Scenario Browser: Adding a New Platform (iv)

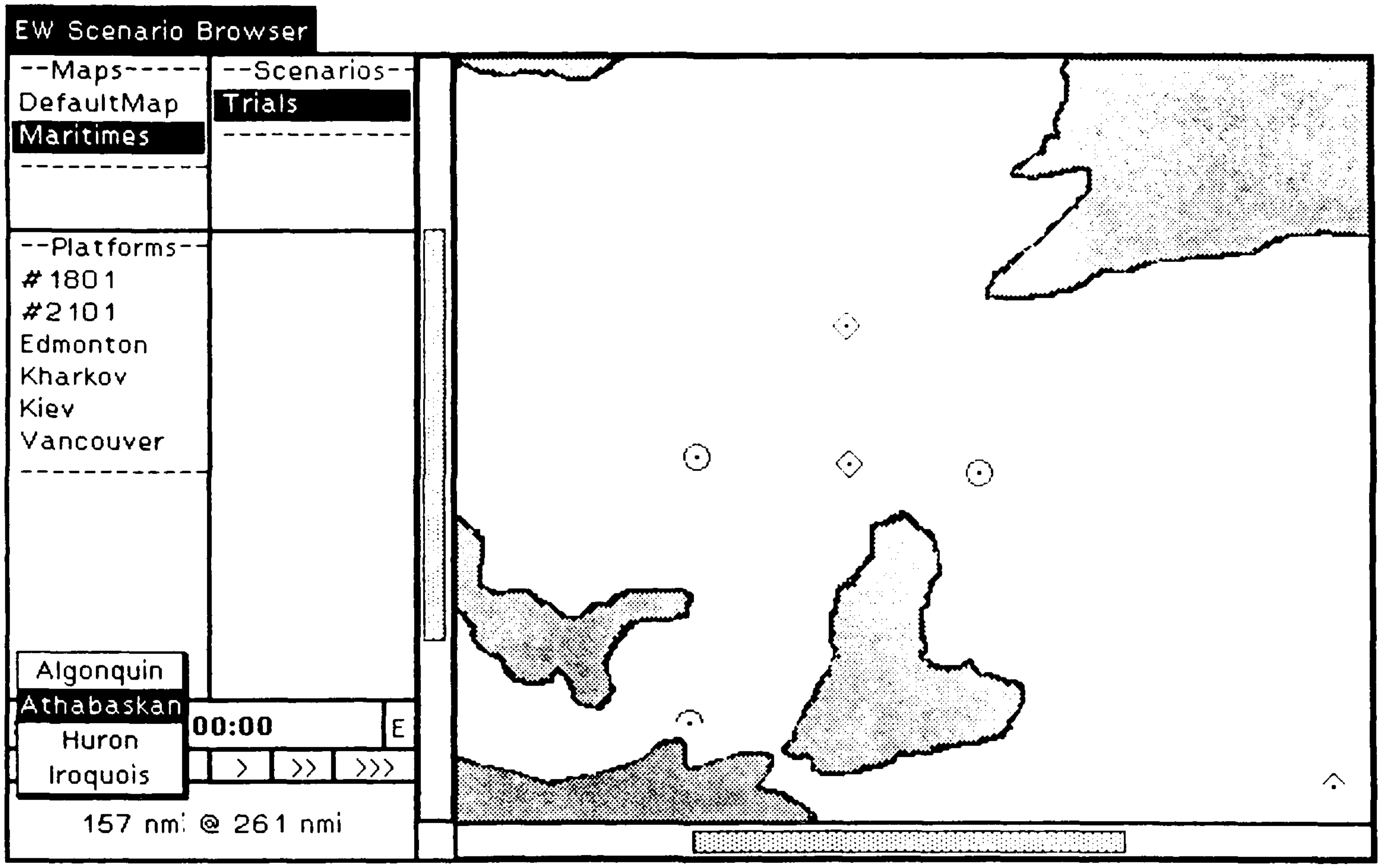
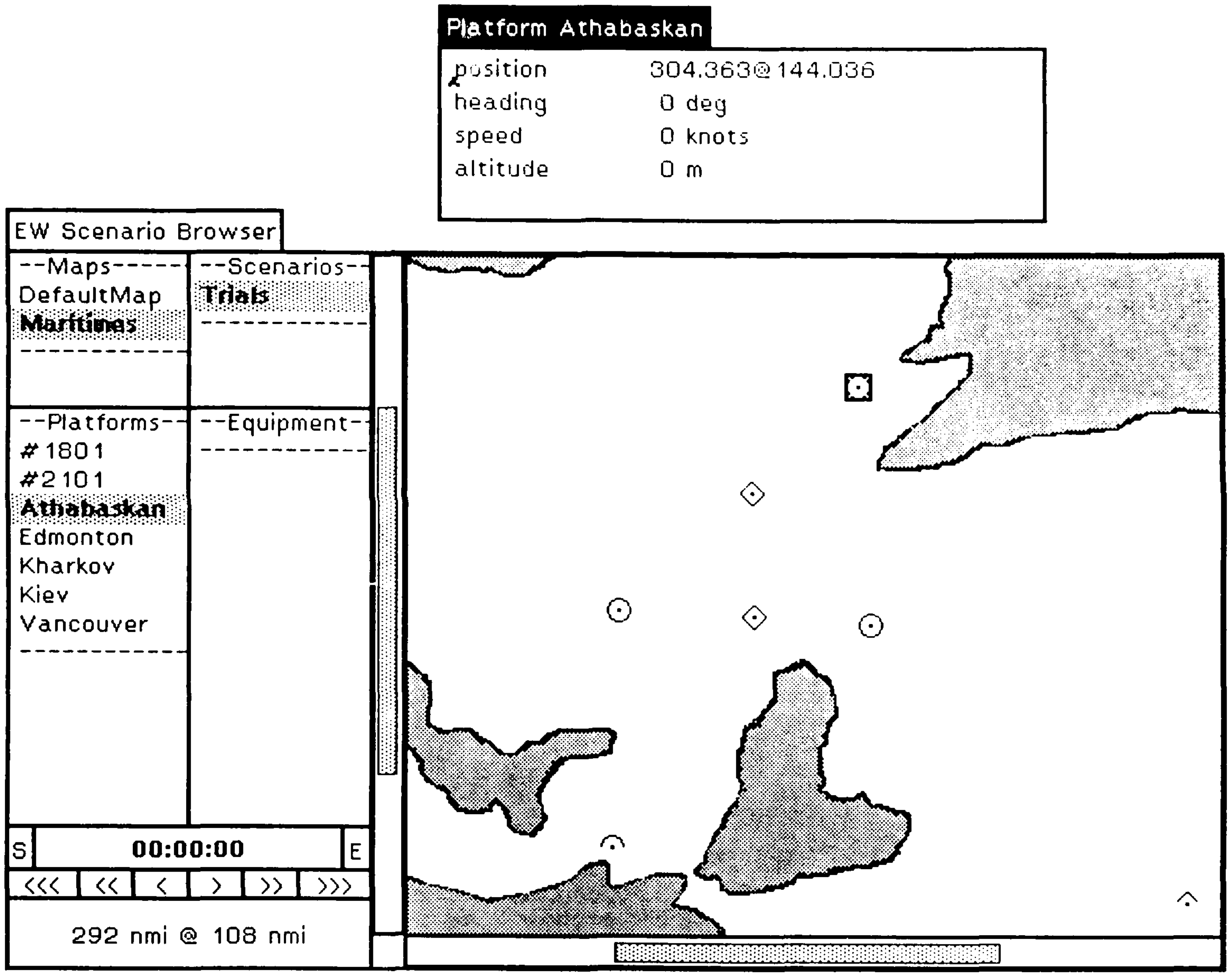


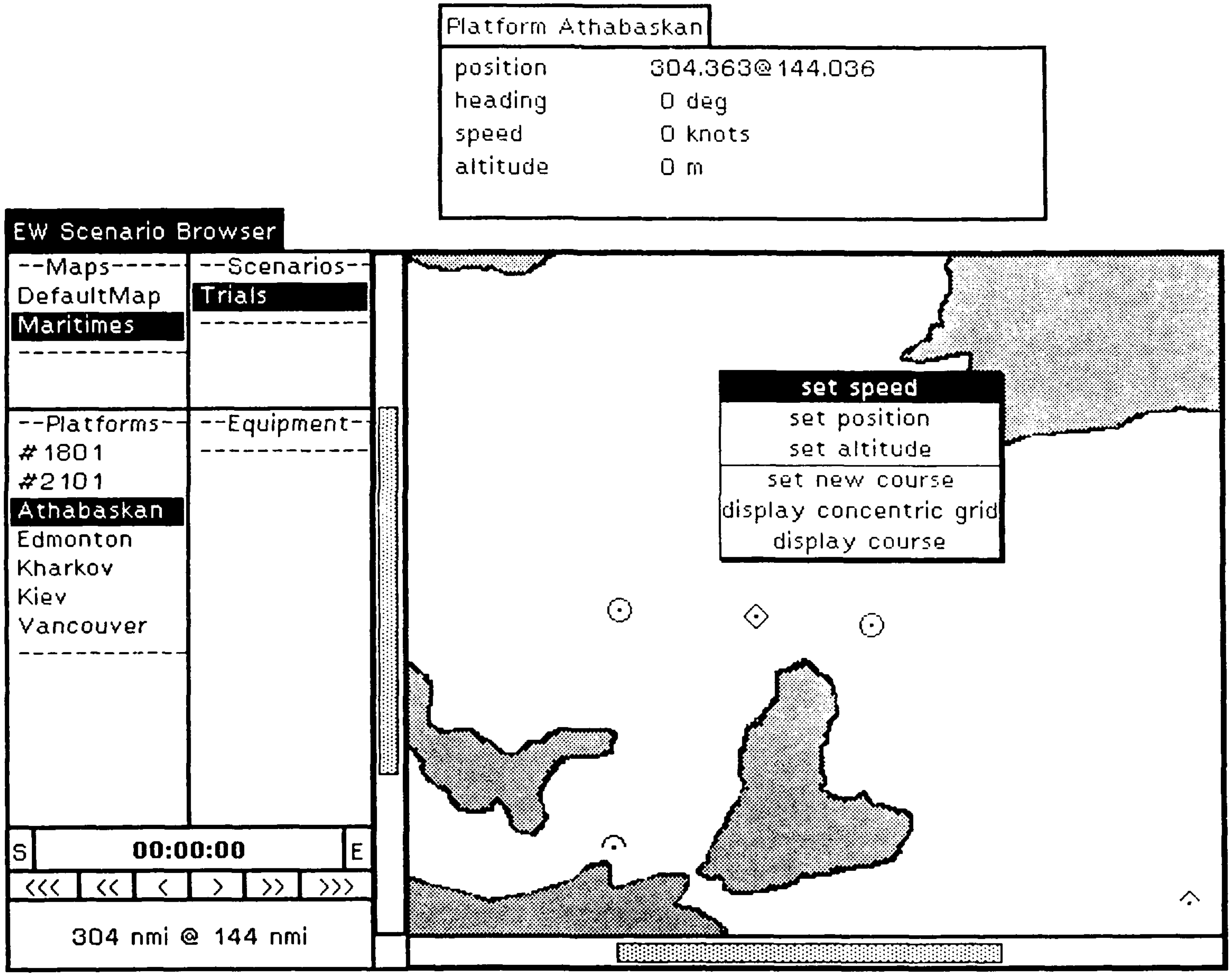
FIGURE 15: Scenario Browser: Adding a New Platform (v)



Platform Athabaskan

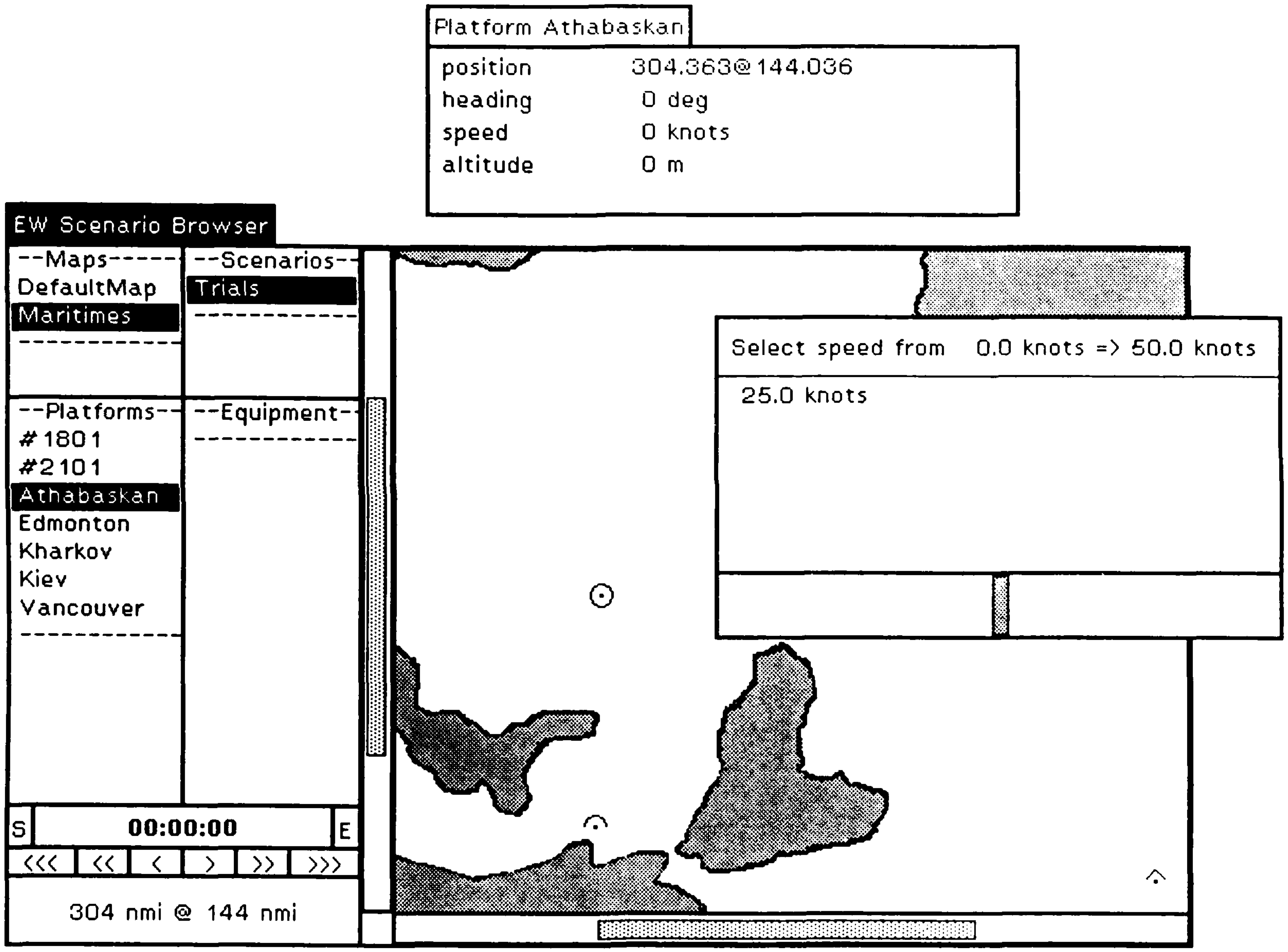
```
position      304.363@144.036
heading        0 deg
speed          0 knots
altitude       0 m
```

FIGURE 16: Scenario Browser: Adding a New Platform (vi)


Platform Athabaskan

position 304.363@144.036
heading 0 deg
speed 0 knots
altitude 0 m

FIGURE 17: Scenario Browser: Setting Platform Speed (i)


Platform Athabaskan

position	304.363@144.036
heading	0 deg
speed	0 knots
altitude	0 m

Select speed from 0.0 knots => 50.0 knots

25.0 knots

FIGURE 18: Scenario Browser: Setting Platform Speed (ii)

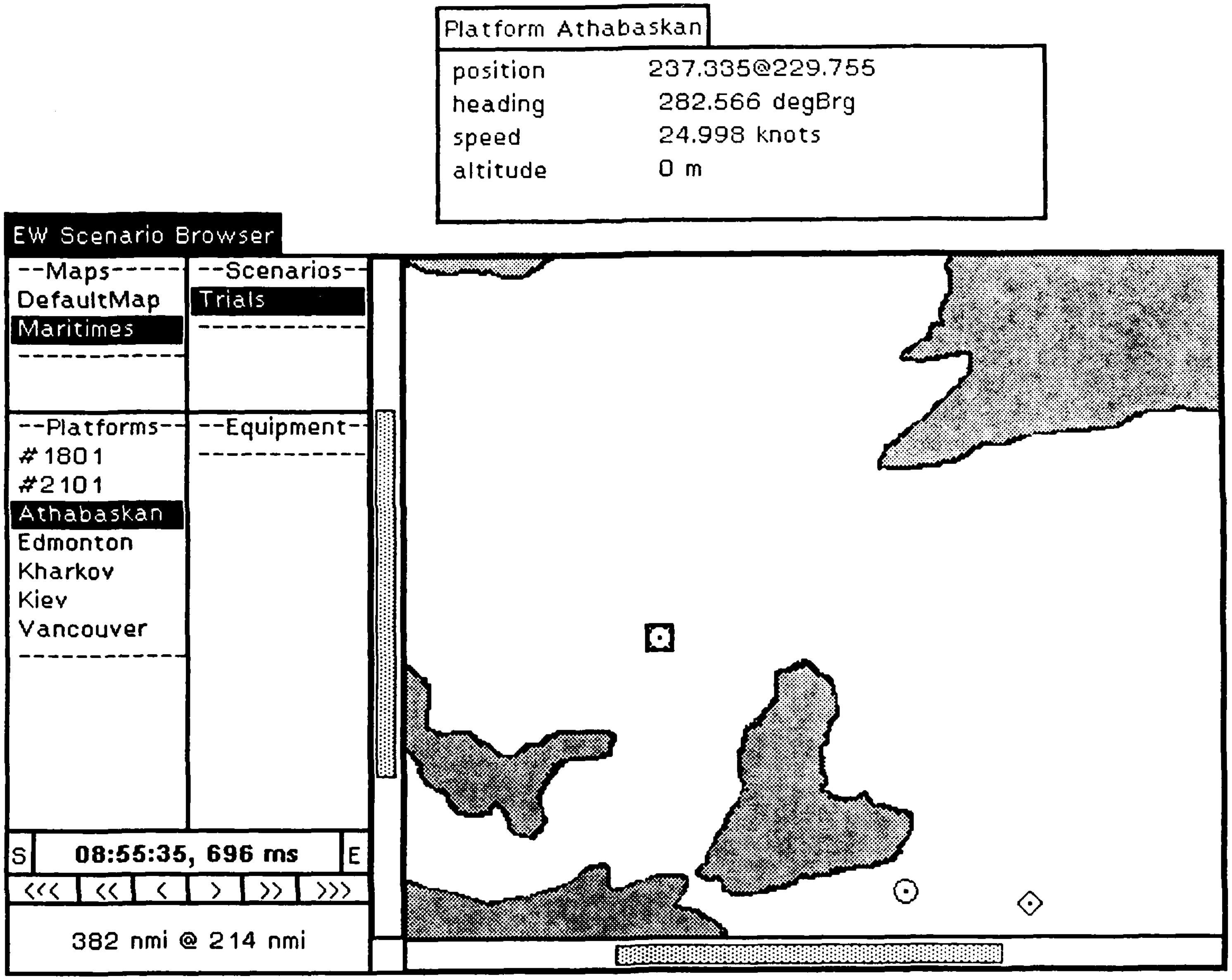
The Scenario Browser displays a clock pane near the lower left corner of the Browser. The user adjusts the clock by clicking on the "buttons" below and to the sides of the digital readout. The "S" and "E" buttons set the clock to the "Start" and "End" times; the ">", ">>", and ">>>" buttons move the current time forward by seconds, minutes, and hours, respectively. Similarly, the "<", "<<", and "<<<" buttons move time backward. When the clock is changed, all platform icons in the scenario move to the map position they would occupy at the new time. For example, holding down the left mouse button after selecting one of the clock buttons results in an animation of the scenario.

Course information is recorded as a set of times and positions. As mentioned, the initial position and time are generated when the platform is created. New course information is added by selecting the "position" menu option in the map pane. The next position is then specified by moving the platform icon to a new point on the map. After the icon has been moved, the Browser automatically computes the elapsed time required for the platform to move to the new position (using the platform speed), and updates the clock. The positions of all other platforms are adjusted to reflect the new time (Figure 19). Selecting the "display course" menu item causes the course of the platform over the entire simulation game to be displayed on the map (Figure 20).

To add emitters to a platform in the scenario, the user first selects the platform. He then moves the cursor to the equipment pane and selects the "add emitter" menu option (Figure 21). This results in a pop-up menu showing a list of the emitter types associated with this platform (Figure 22). After the new emitter is created, its entry is added to the equipment list for the selected platform (Figure 23).

Emitter objects are represented as a list of emitter modes. For each emitter mode, the Scenario Browser's platform/emitter database provides ranges of acceptable values for emitter parameters. When the emitter mode is created, a random value in the acceptable range is selected. In order to change this value, the user selects the "inspect modes" option in the equipment pane (Figure 24).

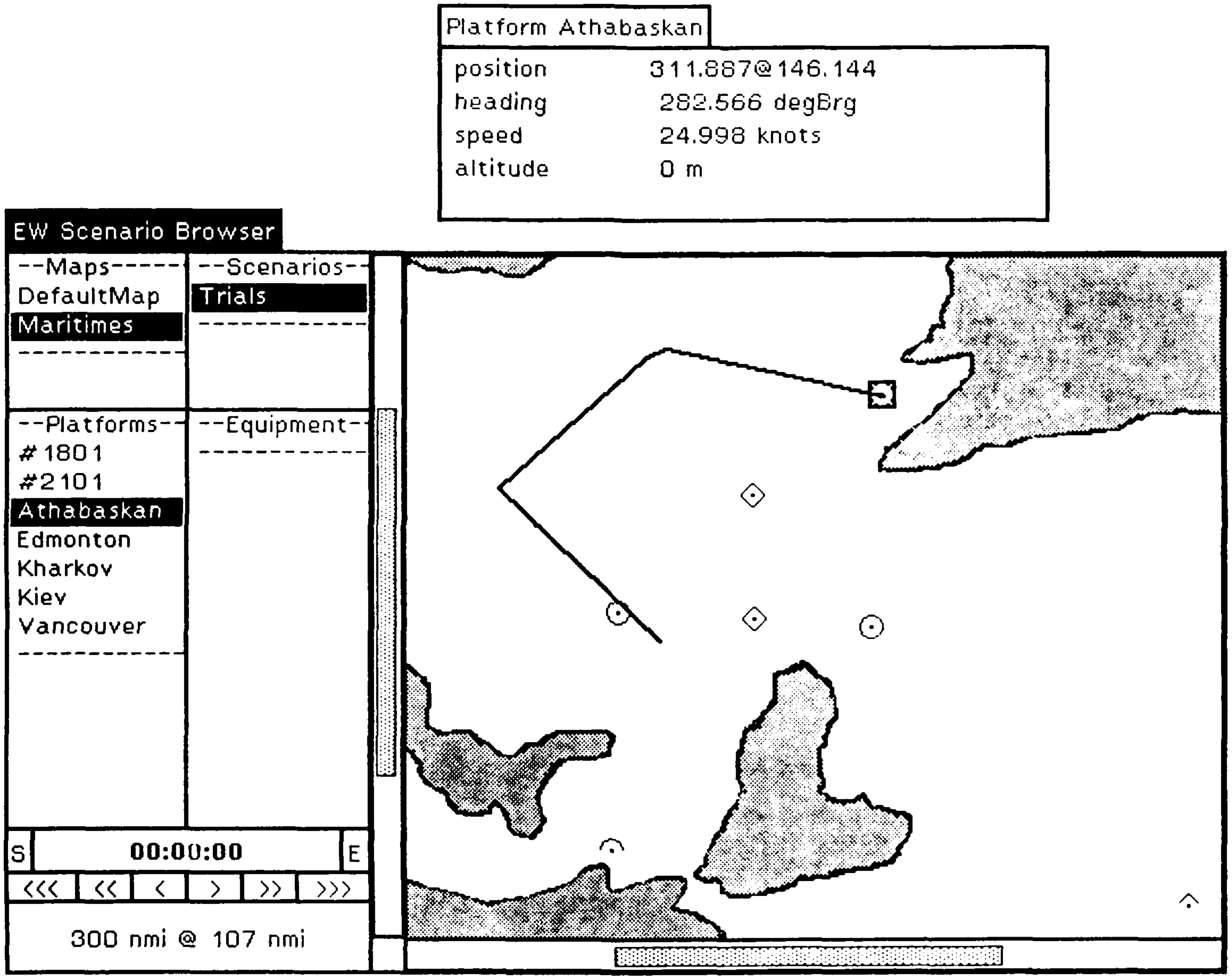
Inspectors are standard Smalltalk objects designed to allow the current state of an arbitrary object to be examined by the user. A default Inspector is provided in the Smalltalk system. It is also possible to write Inspectors which have been customized to examine particular kinds of objects more effectively. As can be seen in Figure 25, an Inspector window consists of two panes: a list of the object's state variables on the left, and a text pane (which can be edited) on the right. Selecting one of the items in the list causes the current value of the selected state variable to be displayed on the text pane. For example, in the figure the mode's "pri" has been selected, and the initial pri value chosen by the Browser is displayed in the text pane.



Platform Athabaskan

position	237.335@229.755
heading	282.566 degBrg
speed	24.998 knots
altitude	0 m

FIGURE 19: Scenario Browser: Updating the Clock and Platform Positions



Platform Athabaskan

```
position        311.887@146.144
heading         282.566 degBrg
speed           24.998 knots
altitude        0 m
```

FIGURE 20: Scenario Browser: Displaying a Platform Course

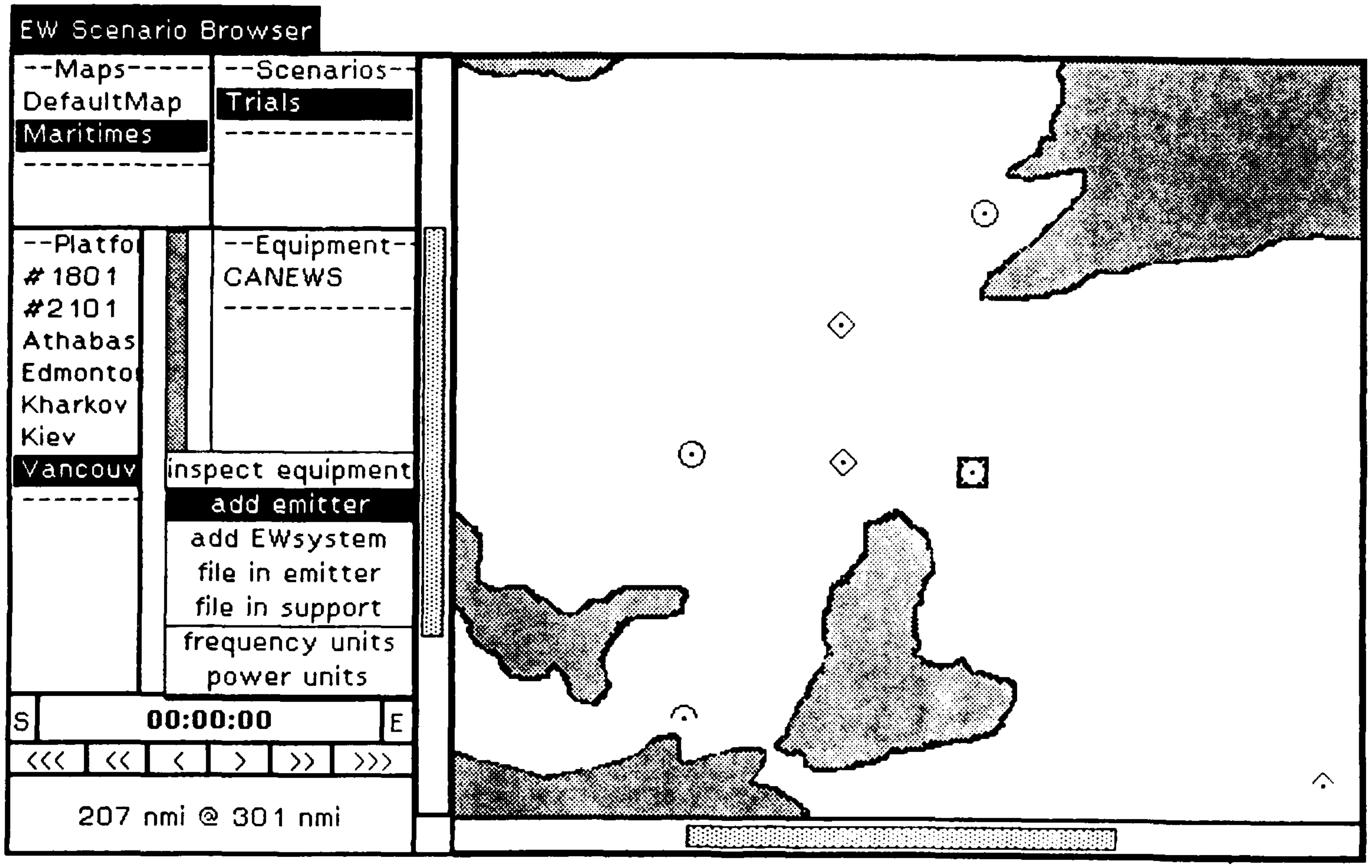


FIGURE 21: Scenario Browser: Adding a New Emitter (i)

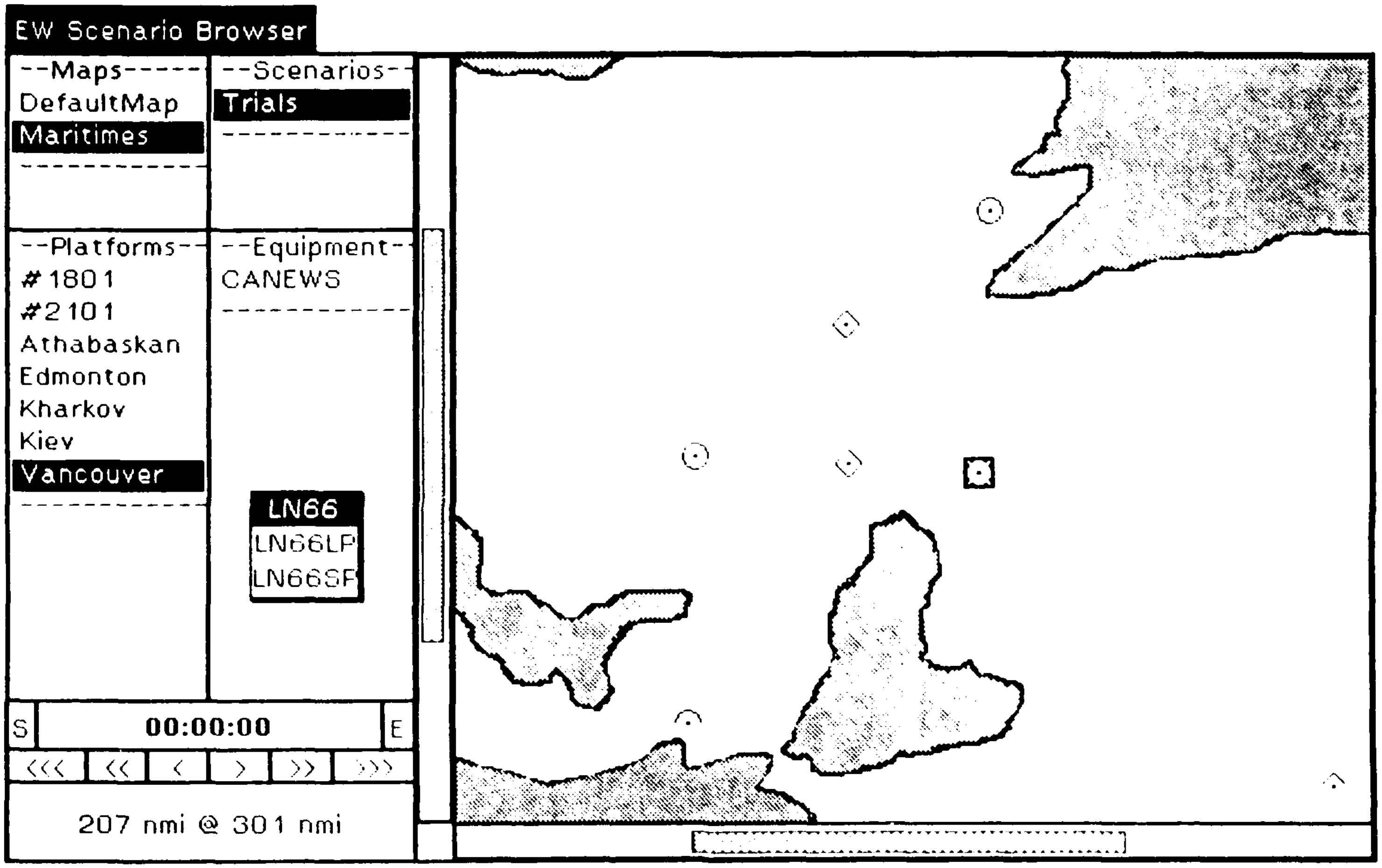


FIGURE 22: Scenario Browser: Adding a New Emitter (ii)

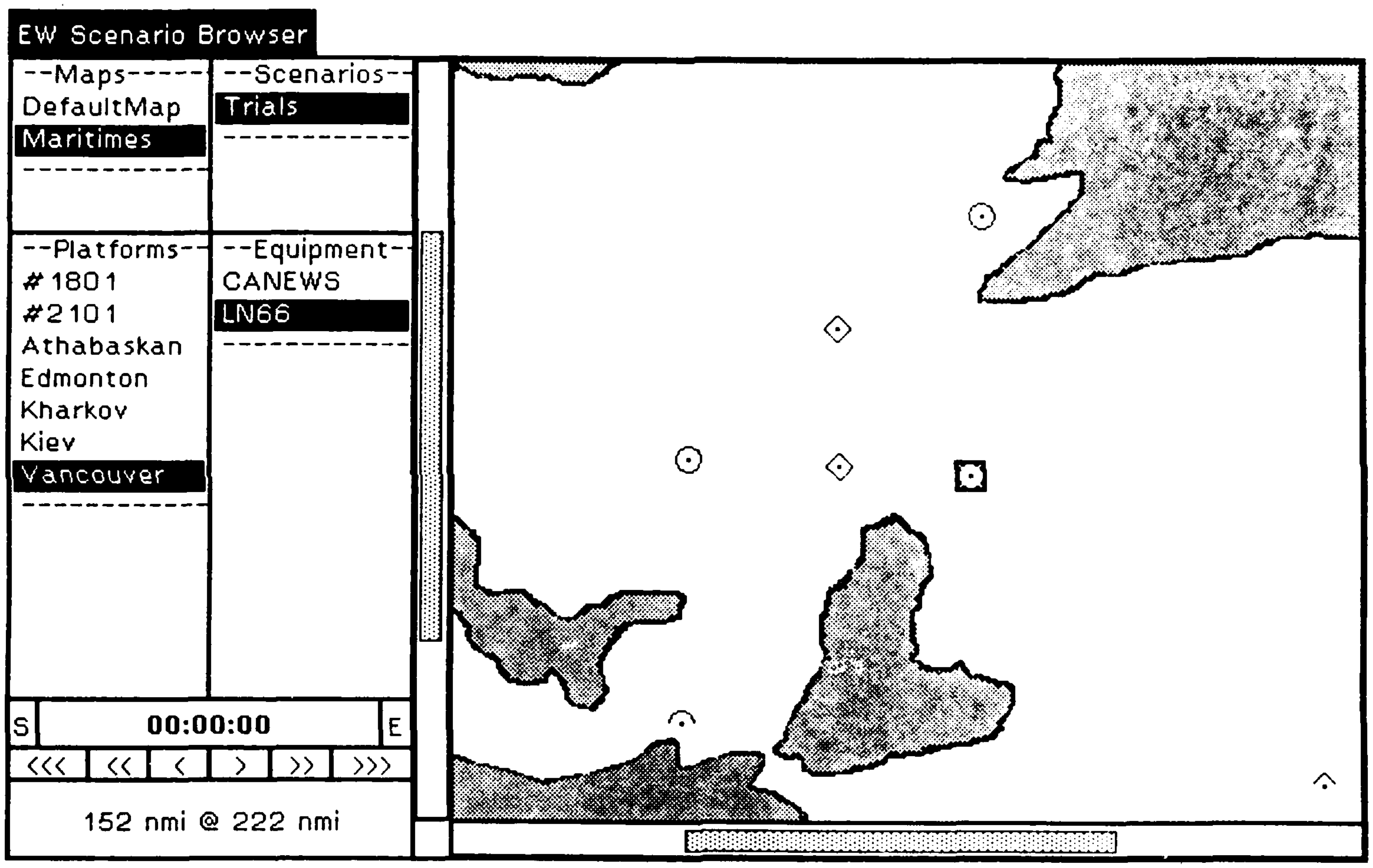


FIGURE 23: Scenario Browser: Adding a New Emitter (iii)

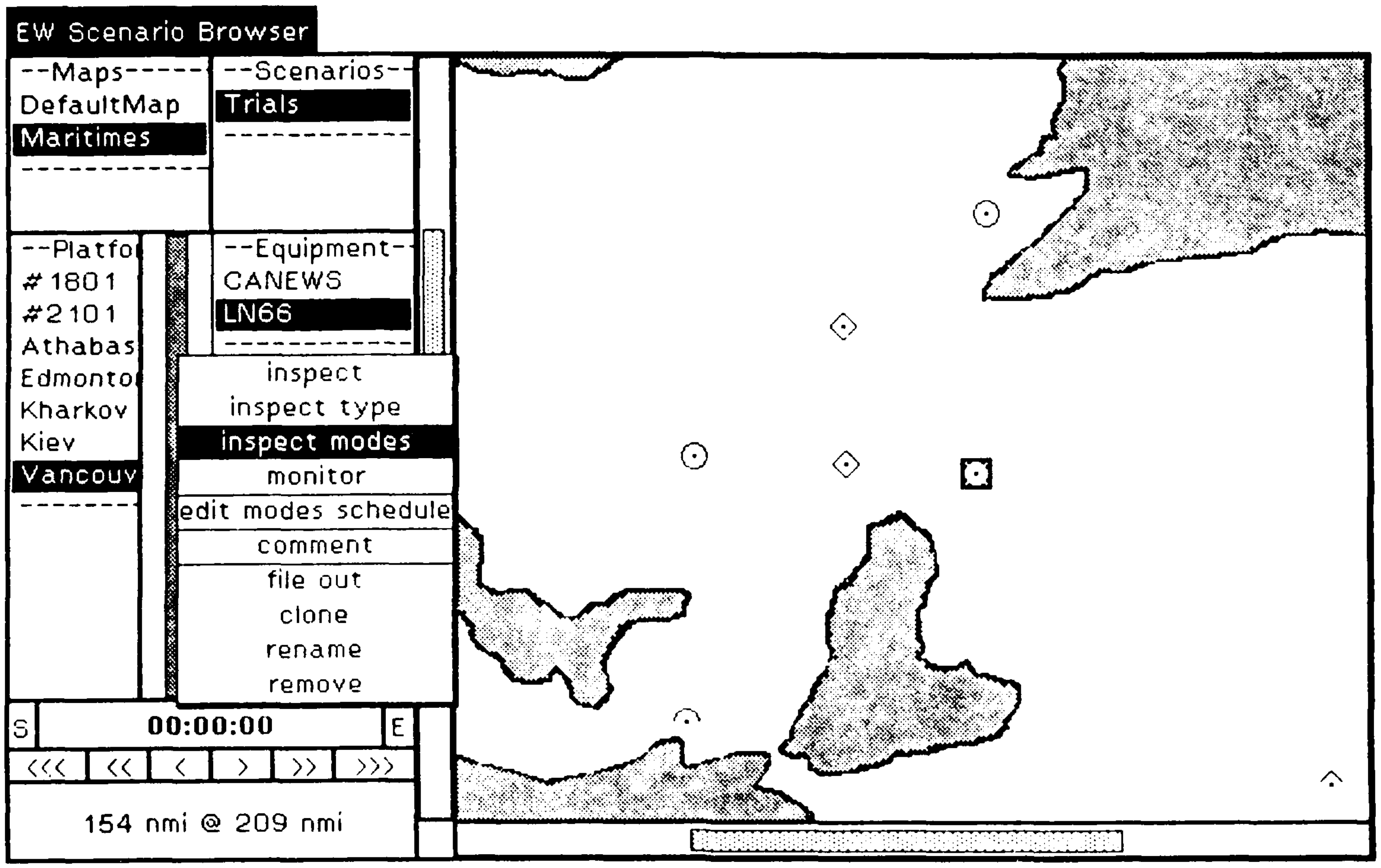


FIGURE 24: Scenario Browser: Inspecting Emitter Modes (i)

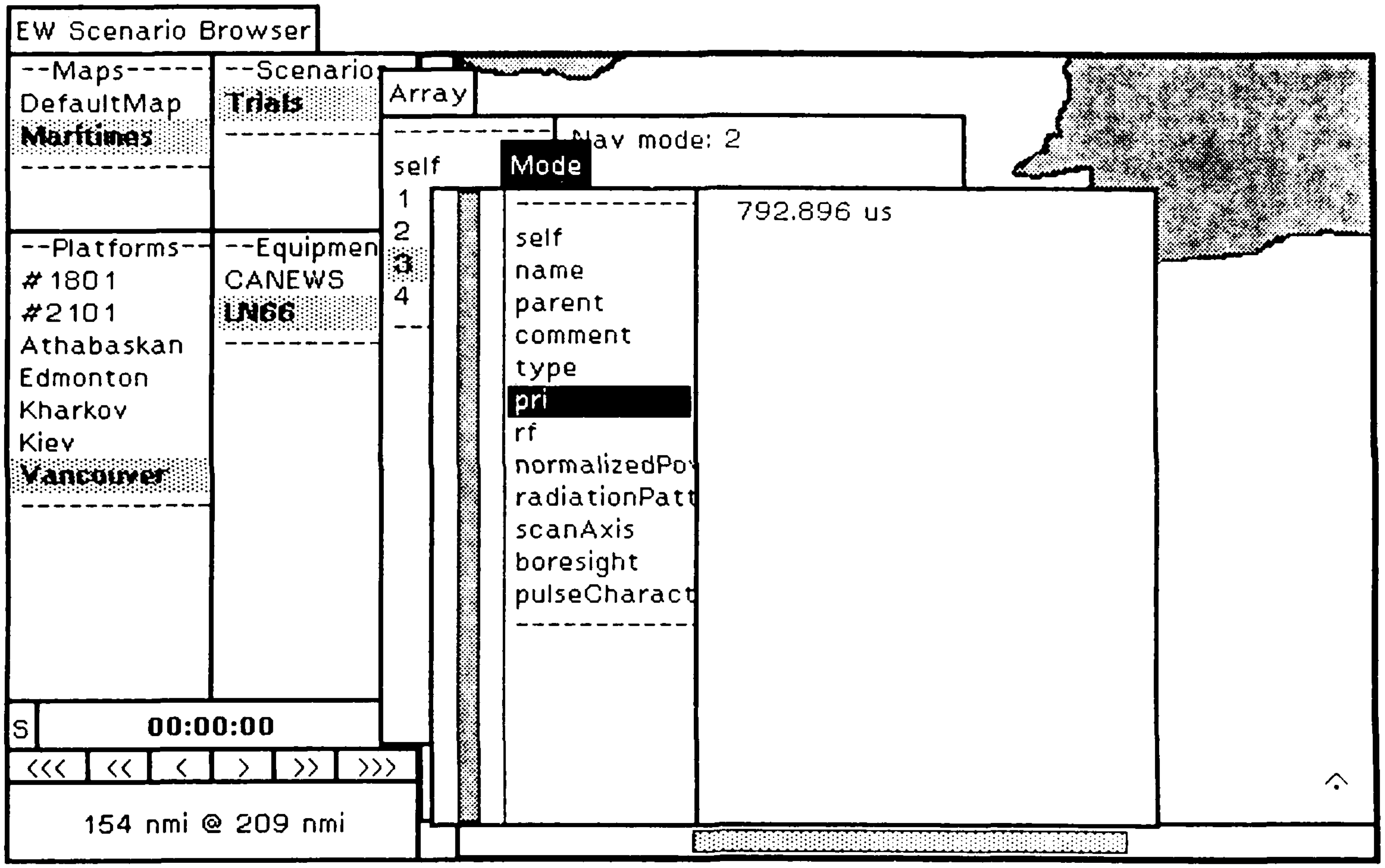


FIGURE 25: Scenario Browser: Inspecting Emitter Modes (ii)

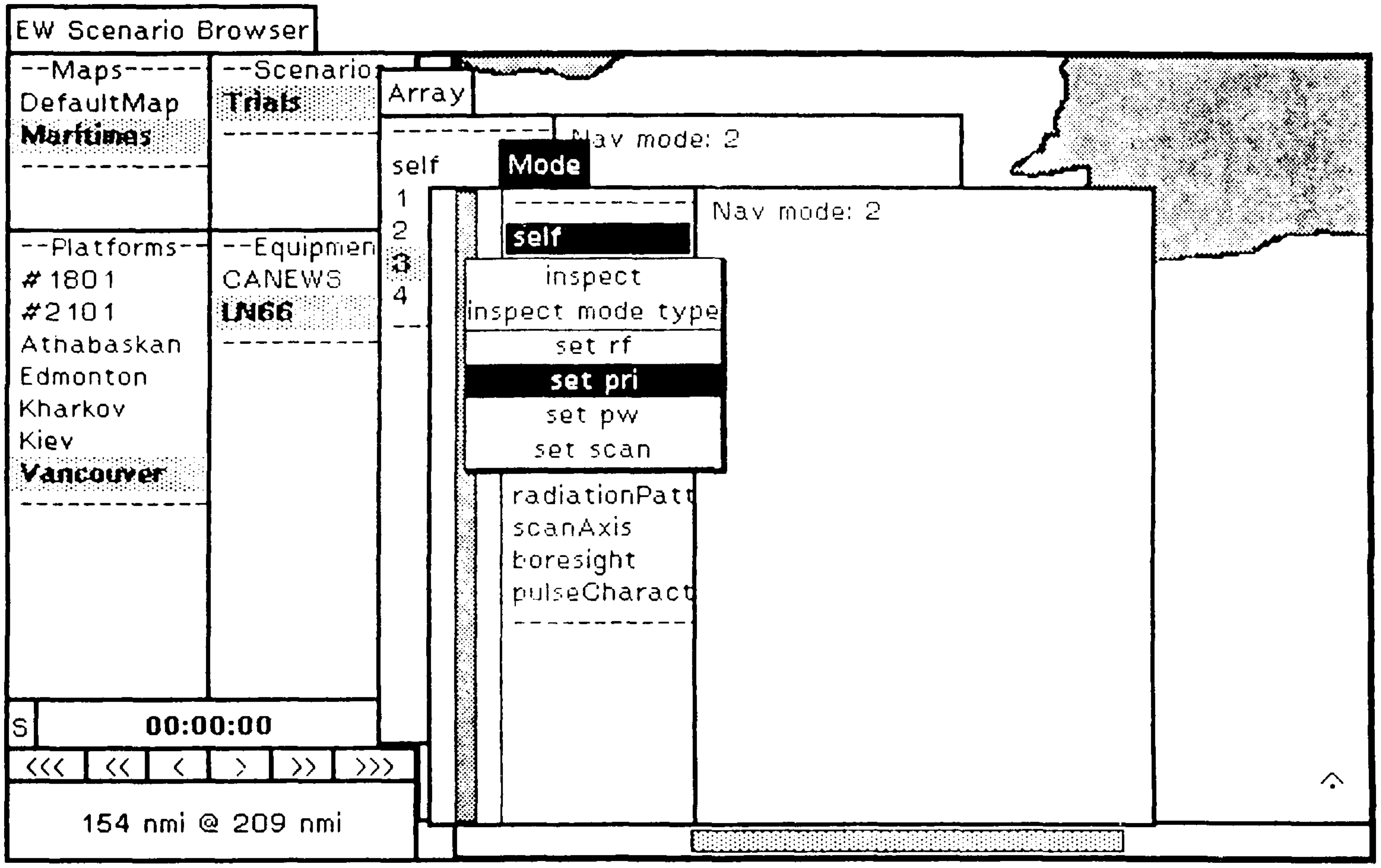
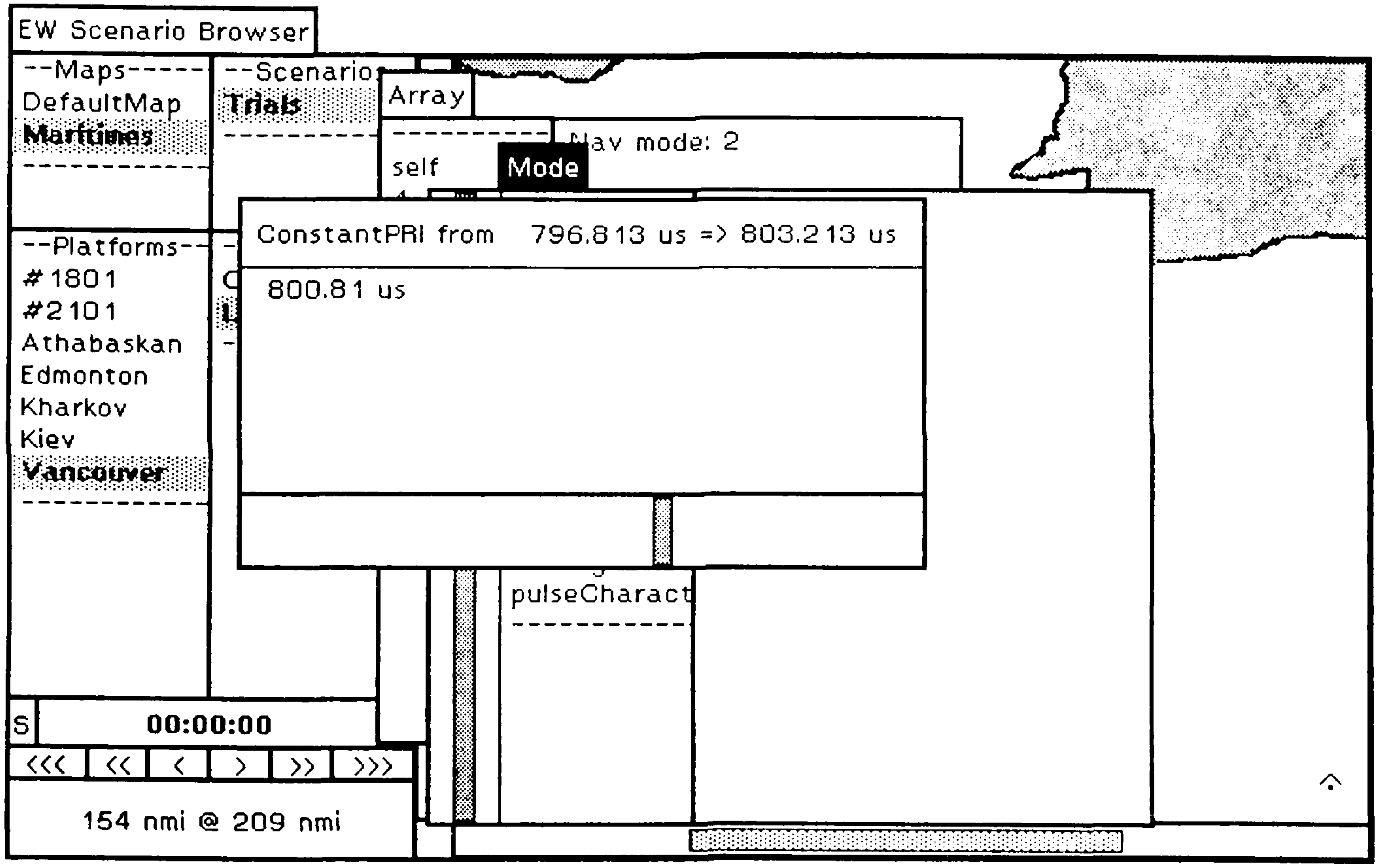


FIGURE 26: Scenario Browser: Changing a Mode Parameter (i)


EW Scenario Browser

--Maps--
DefaultMap
Maritimes

--Scenario
Trials

--Platforms--
#1801
#2101
Athabaskan
Edmonton
Kharkov
Kiev
Vancouver

Array
self
Mode
Nav mode: 2
pulseCharact

ConstantPRI from 796.813 us => 803.213 us
800.81 us

S 00:00:00
<<< << < > >> >>>
154 nmi @ 209 nmi

FIGURE 27· Scenario Browser: Changing a Mode Parameter (ii)

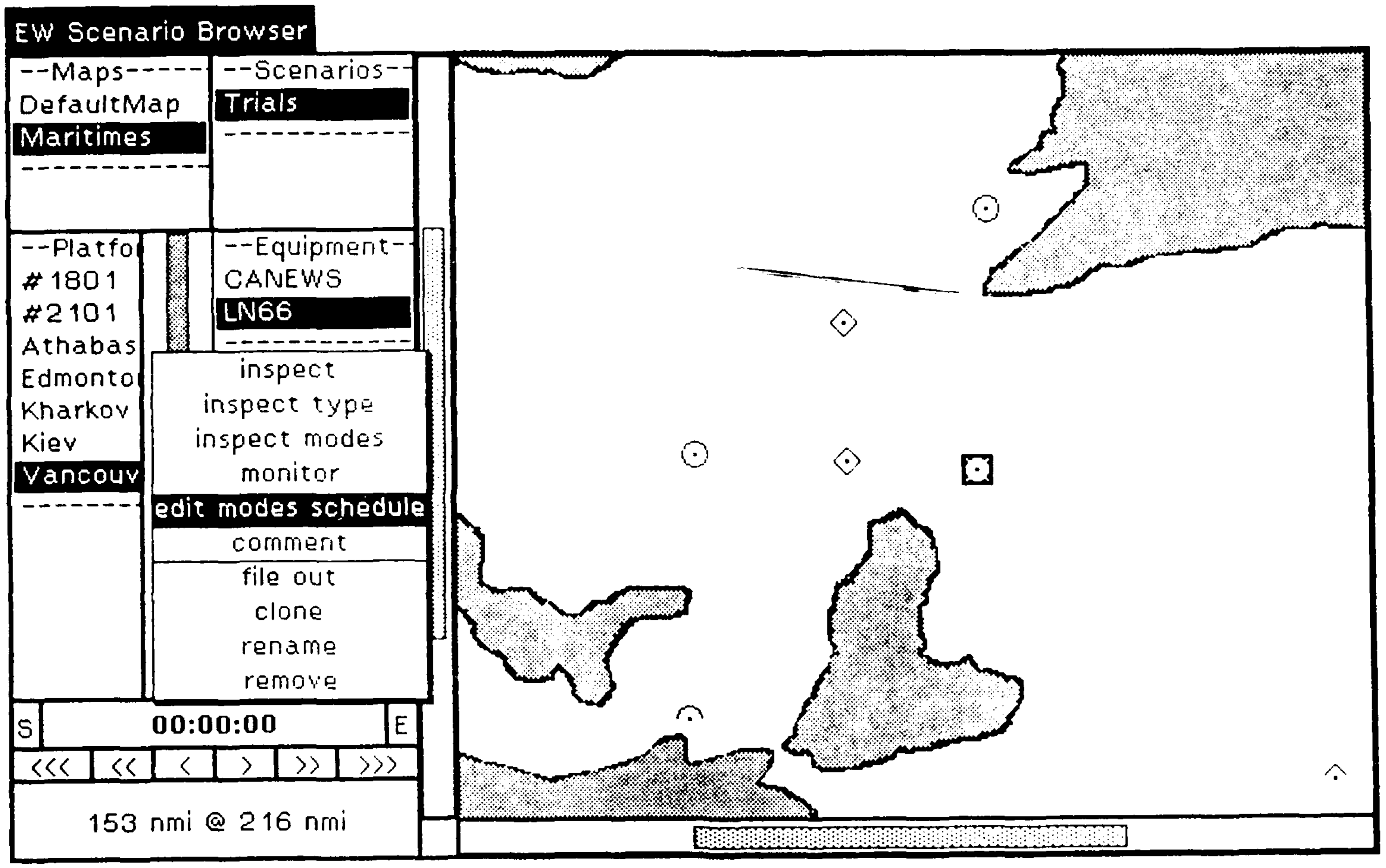


FIGURE 28: Scenario Browser: Editing the Modes Schedule (i)

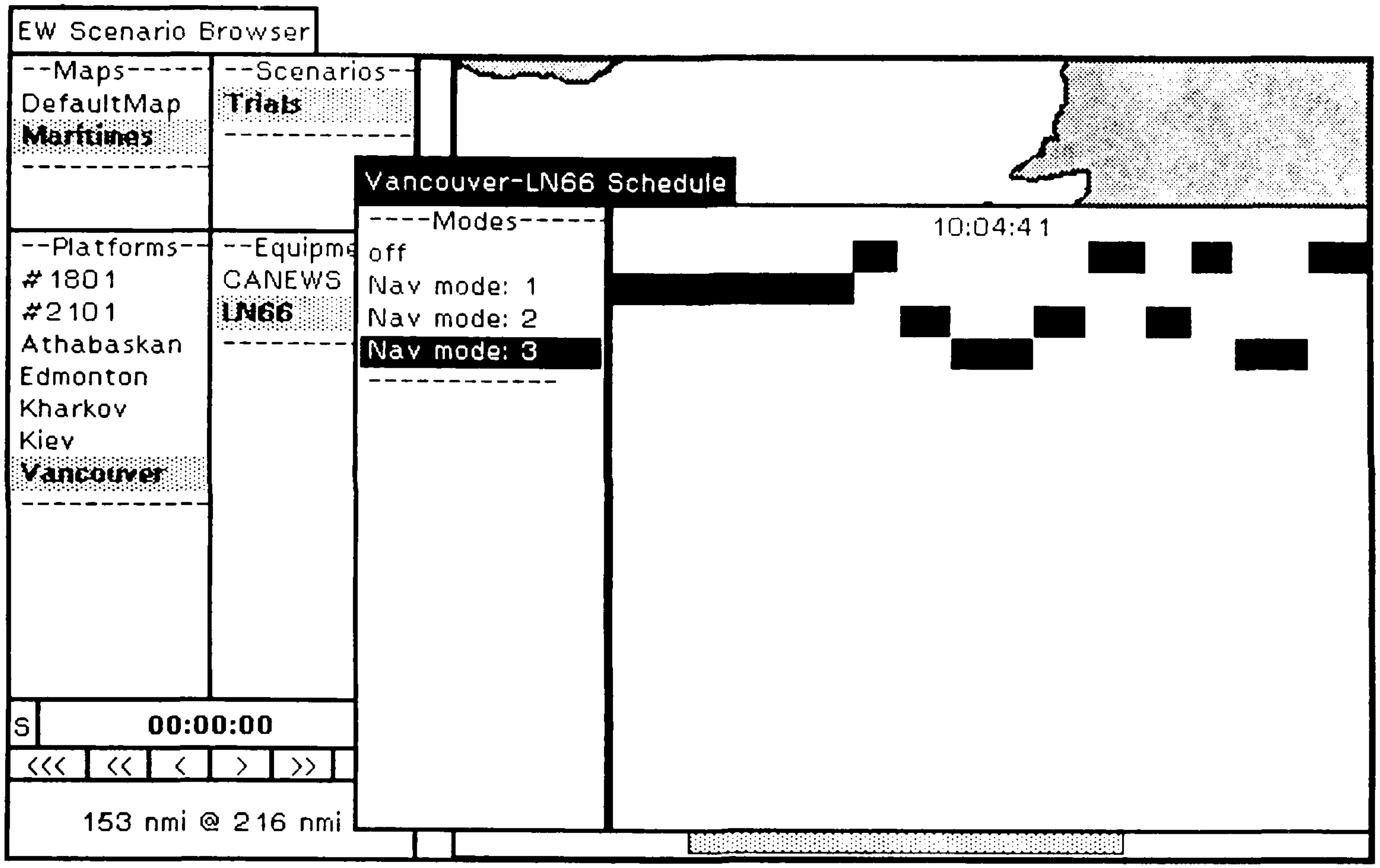


FIGURE 29: Scenario Browser: Editing the Modes Schedule (ii)

As shown in Figure 26, menu options are available in the Inspector which allow the user to change the parameters of emitter modes. For example, selecting the "set pri" option results in the limited choice menu shown in Figure 27. Once again, the browser prompts the user to select a pri value from within an acceptable range. Selecting the "edit modes schedule" option in the equipment pane (Figure 28) results in a pop-up schedule view which shows a time line for each mode of the emitter (Figure 29). These time lines specify the operating mode being used during each time interval the emitter is actually radiating. Simple mouse input is all that is required to change the modes schedule.

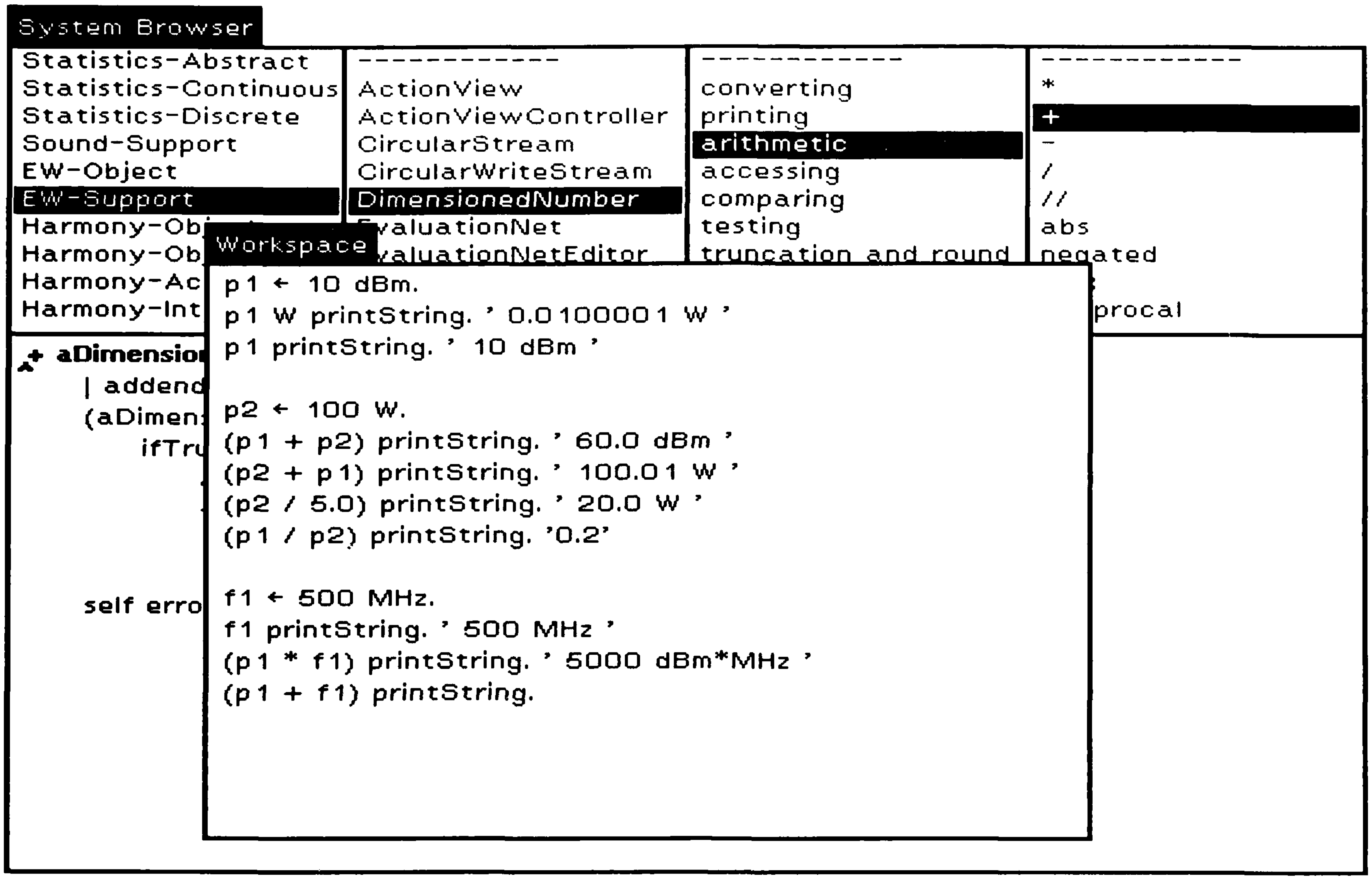
3.4 Dimensional Analysis

Some of the most annoying and persistent problems encountered when implementing simulations are the result of arithmetic operations which are dimensionally inconsistent, e.g. trying to compute a distance by multiplying speed measured in miles per hour and time measured in seconds. Indeed, many experts counsel that dimensional analysis is a good "rule of thumb" check on the validity of the mathematical formulae used in a simulation. Parameters associated with Platforms and Emitters in a Scenario always explicitly display their units of measurement. This is not cosmetic, but actually reflects the fact that they are represented as Dimensioned Numbers which store both a magnitude and units of measurement. Arithmetic operations performed on Dimensioned Numbers check the units of measurement and generate an error when dimensionally inconsistent operations are invoked. In this way, we can implicitly perform dimensional analysis on every computation done during a simulation. To simplify the user interface, protocol has been added to the Smalltalk class Number to support English language-like creation of Dimensioned Numbers.

Figures 30 and 31 show a short Smalltalk work session which demonstrates the use of Dimensioned Numbers. We begin by typing

```
pl ← 10 dBm.
```

This creates a Dimensioned Number object with power units. In the current implementation, Dimensioned Numbers support a limited number of conversion, comparison, and arithmetic operations (addition, subtraction, multiplication, and division) directly. For more involved computations, a Dimensioned Number can provide a scalar quantity scaled to the appropriate units on request. For example, as shown in Figure 30, we can send pl the message "W printString" ("printString" generates a character string representing an object), which returns the value in units of watts. Alternatively, we can just ask pl to print itself, in which case it returns 10 dBm.



```
+ aDimensio
    | addend
    (aDimen
        ifTr

    self erro
```

```
p1 ← 10 dBm.
p1 W printString. ' 0.0100001 W '
p1 printString. ' 10 dBm '

p2 ← 100 W.
(p1 + p2) printString. ' 60.0 dBm '
(p2 + p1) printString. ' 100.01 W '
(p2 / 5.0) printString. ' 20.0 W '
(p1 / p2) printString. '0.2'

f1 ← 500 MHz.
f1 printString. ' 500 MHz '
(p1 * f1) printString. ' 5000 dBm*MHz '
(p1 + f1) printString.
```

FIGURE 30: Smalltalk Work Session: Illustrating Dimensioned Numbers

The next example in the figure illustrates how Dimensioned Numbers implement dimensional analysis. We first create two power objects pl and p2. Notice that when we add pl and p2, the result is displayed with correct units (the units of the first addend are used as the default). Similarly, when we divide p2 by a scalar quantity ("p2/5.0") we retain the power units. However, dividing pl by p2 results in a scalar (note that no units are printed), just as it should. Moreover, computations which do not make sense from a dimensional standpoint are trapped. In the next line, we create a Dimensioned Number with frequency units, fl. When we multiply pl and fl, we obtain the correct result. However, Figure 31 shows the result when we try to add fl and pl. The system displays a "Notifier" window with the error message: "Conversion not available".

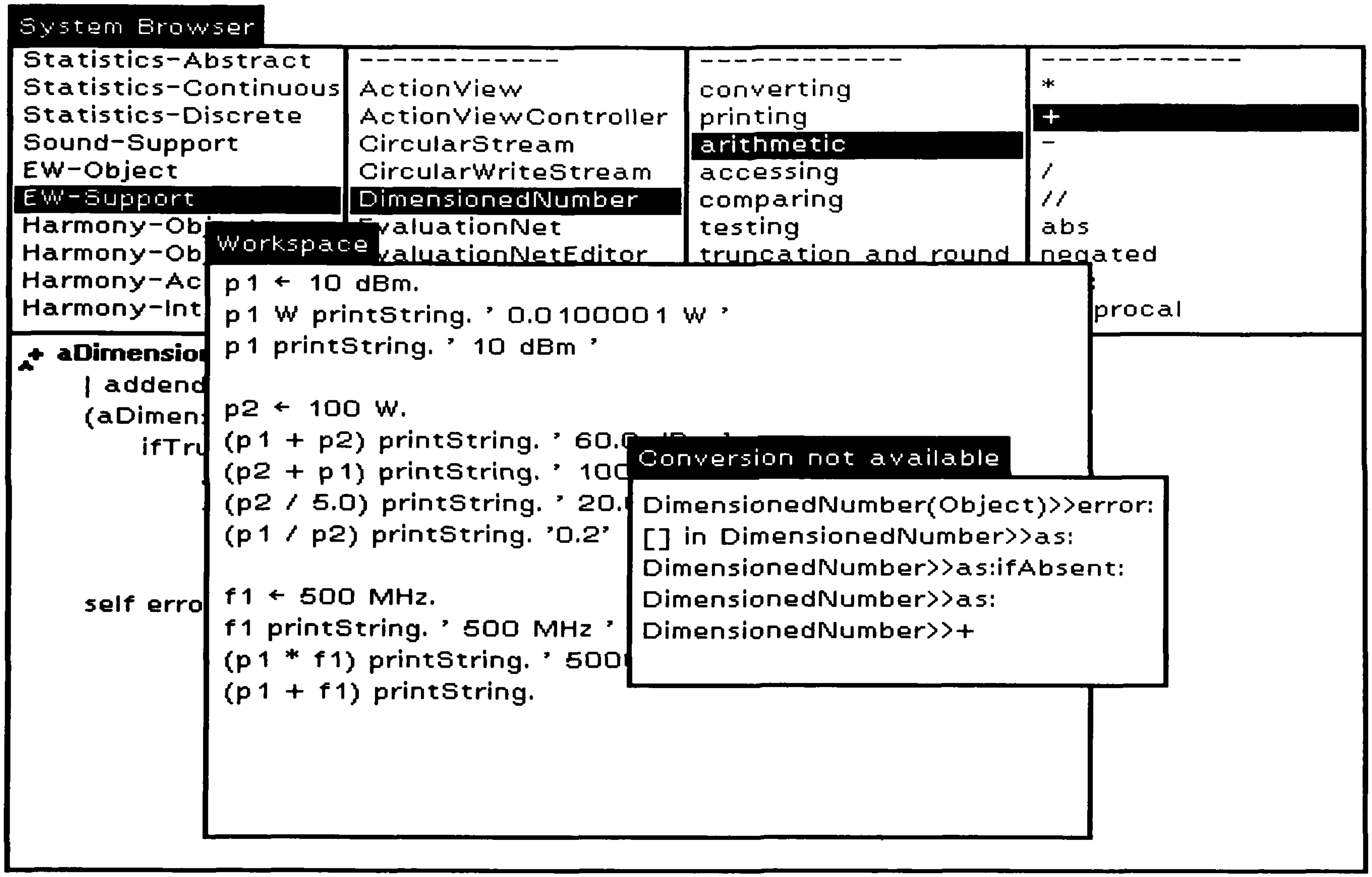
If for some reason a user actually had wanted to add a Power and a Frequency, the system doesn't really prevent this. By sending explicit access messages (e.g. "pl W value", "fl MHz value") to the objects one can obtain dimensionless quantities which can then be added. However, this requires an explicit choice on the part of the user. The intent of this system is provide the end-user with a flexible mechanism which eliminates errors due to carelessness or bad design. It should also be remarked that there is obviously a performance penalty imposed by the additional message sends required to calculate with Dimensioned Numbers. However, real performance is measured by the overall effectiveness of the simulation as a tool for understanding the modelled system's behaviour. In our view, having programmers spend weeks trying to locate an error caused by mismatched dimensions, or having a simulation which is simply wrong, has a more significant impact on performance.

4.0 REFLECTIONS ON LESSONS LEARNED AND THE WAY AHEAD

The prototype simulation environment described in the last section was essentially designed and implemented over a six month period on a Tektronix 4404 AI workstation. About 8 man-months effort were required, divided roughly into three man-months for design and five man-months for the partial implementation of that design. Both the designer and the implementer had some experience programming in Smalltalk but were by no measure Smalltalk experts, so that at least some of their time was spent learning more about Smalltalk and object-oriented programming methods. Despite this, the results have surpassed our expectations. The functionality provided by the programming environment compares favourably to other EW simulation systems which were developed at significantly higher costs.

4.1 Assessment of the Prototype Simulation Environment

Since the prototype was developed to demonstrate that many of the limitations of existing EW simulations could be remedied, it seems fair to evaluate the extent to which it has addressed these concerns. Table 1 shows the list of requirements developed in Section 2.2, and indicates on assessment as to how well the prototype simulation environment meets the requirements. The most obvious success is the degree to which object-oriented simulation preserves the relationships which exist between modelled objects in the real world. Since this was the main motivation behind adopting the object-oriented approach, it was satisfying to see our hopes realized.



```
aDimensio
    | addend
    (aDimen
        ifTru

    self erro
```

```
p1 ← 10 dBm.
p1 W printString. ' 0.0100001 W '
p1 printString. ' 10 dBm '

p2 ← 100 W.
(p1 + p2) printString. ' 60.0
(p2 + p1) printString. ' 100
(p2 / 5.0) printString. ' 20.
(p1 / p2) printString. '0.2'

f1 ← 500 MHz.
f1 printString. ' 500 MHz '
(p1 * f1) printString. ' 500
(p1 + f1) printString.
```

```
DimensionedNumber(Object)>>error:
[] in DimensionedNumber>>as:
DimensionedNumber>>as:ifAbsent:
DimensionedNumber>>as:
DimensionedNumber>>+
```

FIGURE 31: Smalltalk Work Session: Attempting a Dimensionally Inconsistent Operation

TABLE 1: Assessment of Prototype Simulation Environment

Requirement	Good	Fair	Poor	None
Open, articulate system; supports multiple views		X		
Direct correspondence between model and real world systems	X			
Uses standard interfaces; automatic software version management		X		
Explicit support for validation and verification			X	
Built-in event logging and analysis without code modification		X		
Easy access for non-programmers	X			

We have given the environment a "Fair" score in the three categories which deal with software system architecture. However, the only fundamental limitation here was manpower. When developing software systems with object-oriented methods the preferred technique is to make extensive use of existing classes, adding new objects to the system only when necessary. Cox [9] makes the analogy between this style of programming and the way electronics engineers use integrated circuits to design ever larger applications in their domain. At the moment, the only "object library" available is that provided with the Smalltalk language. Applications which can be built mainly from these existing objects can be implemented quickly. Those which require a large number of new objects have a longer development time, as effort must first be put into creating the new objects. The EW simulation environment seems to fall somewhere between the two extremes. We should note that follow-on work is already making use of some of the EW objects created for this applications.

The use of browsers and multiple views in object-oriented systems is fast becoming commonplace. With sufficient effort, it seems clear that the goal of having an open, articulate system can be met. Smalltalk itself is an excellent example of a programming environment which provides automatic configuration management using an object called the System Organizer. Since this is currently being used without modification, it does not distinguish between code generated to implement a particular simulation and code developed to support the simulation environment as a whole. This could be remedied by creating a new Simulation Organizer class. Instances of this class could be created to document each simulation.

The Monitor objects described in Section 3.2 are an indication of the kind of event-logging capability we are seeking. While Monitors do not include any analysis tools at the moment, these could easily be added. The important point to note about Monitors is that they allow the user to inspect and log interesting events associated with Actors without having to modify any of the Actor's procedures. What is missing is some way of characterizing for the system a notion of "interesting event" which would trigger an action such as displaying a Notifier, collecting statistics, or logging an event on a file. This probably requires a better formalism for describing the interaction between objects in message-passing systems than currently exists.

We are still a long way from realizing our goal of providing easy access for non-programmers. However, the potential advantages of the object-oriented approach have been demonstrated through the example of the Scenario Browser, which is a powerful tool easily accessible to non-programmers. However, to achieve this ease of use required a significant proportion of the total time and effort. Since the goal of the prototype development was to prove concepts, it was not feasible to develop more than one full-fledged user interface of this sort.

Explicit support for validation and verification is the one category where not much progress has been made. We rated the prototype as "Poor" rather than "None" primarily because inheritance provides a capability to re-use code which limits programming errors, and the Smalltalk Browser makes code more accessible and maintainable. As indicated in Section 2.2, we suspect that a capability to build models rather than write programs is called for. There are several Smalltalk applications designed for simulating simple physical systems which indicate that such an approach is feasible; these include ThingLab [10], Animus [11] and the Alternate Realities Kit [12]. All of these systems place heavy emphasis on programming by means of graphical descriptions of the relationships between objects.

4.2 Future Developments

A number of improvements in the simulation environment are either planned or ongoing. Now that we have developed a small library of application specific classes which can be modified and re-used, development is proceeding quickly on a number of fronts. For example, an improved description of Emitter objects which was developed as part of a related effort to develop an object-oriented emitter database [13] is being integrated into the environment. This will allow modifications to the Scenario Browser so that standard platforms can be added to a scenario complete with their "normal" complement of Emitters. Other enhancements to the ScenarioBrowser include: a less cluttered display format which makes more extensive use of pop-up windows, a graphical mechanism for defining and displaying scan patterns, and better methods for defining and displaying platform trajectories.

Most EW simulations now in use have been designed to use either "signal level" or "pulse level" descriptions of emitter behaviour. In signal level simulations, intercepts are modelled by calculating the time and frequency coincidences of the emitted radar signals and the EW system's receiver. At the simulated time of coincidence, signal amplitude is computed or retrieved from a look-up table. It is then compared with the receiver's sensitivity to determine whether or not the "intercepted" signal is above the EW system's detection threshold. If it is, then the EW signal processing which would be applied to the signal is simulated. No pulse trains are generated, and no pulse-level signal processing is simulated. In effect, signal-level simulations simulate time-averaged behaviour over the period of the intercept. For example, if two or more signals are intercepted in the same frequency/time window, pulse train deinterleaving would be simulated by a set of predetermined rules.

Pulse level simulations involve detailed modelling of both pulse trains and EW signal processing functions. Pulse level simulations seem to have more credibility but are limited to simulating a small number of emitters to avoid computational overload. Signal level simulations can simulate dense signal environments but sacrifice fidelity. Pulse Packet and Pulse Train objects are being added to the environment to provide a better capability to describe signals. Pulse Packets are collections of pulses from a single source (e.g. the pulses which might be received during illumination by the main beam of a scanning radar). Pulse Trains are collections of pulse packets. These objects will allow us to simulate emitters at the signal level. However, by providing Pulse Packets with procedures for generating individual pulses, we will retain a simultaneous capability to simulate signal processing at the pulse level.

As discussed previously, the underlying structure of Receiver objects was designed with the idea of allowing a systems engineer to define EW receiver models directly with system block diagrams. In order to realize this capability, it would be necessary to implement the Receiver Browser described in Section 3. The Receiver model could be further improved by providing a capability to automatically "compile" the network description of the Receiver into code. Although the Processor and Task objects used to model ESM signal processors have proven to be quite versatile and powerful, several optimizations are possible in this area as well. The development of better techniques for debugging simulations of multi-tasking systems is a difficult and challenging problem. Indirection Objects which allowed messages to be re-routed transparently would simplify the design of Monitors [14]. Finally, incorporating the features of the Actra [5] system into the simulation environment would lead to a much more elegant implementation of Actor objects.

5.0 CONCLUSION

In summary, we have described a prototype object-oriented EW simulation environment which demonstrates that many of the limitations of traditional EW simulations can be remedied. This should not be surprising. As Cox [9] points out, object-oriented methods provide powerful tools which focus on system-building rather than program-building. Without such tools simulation designers are unlikely to produce results which are both timely and credible. Most of the problems with traditional simulations which were described in Section 2 are structural weaknesses resulting from an inability to manage complexity at the systems level. While the prototype simulation environment we have described is by no means a panacea, it is a definite step in the right direction. Object-oriented techniques result in simulations which are easier to develop, modify, validate, and use. We believe they hold the promise of providing shorter development cycles, more understandable systems, and more credible end products.

6.0 REFERENCES

[1] J.A. Altoft, Brian M. Barry, and Robert Inkol, "A Parallel Architecture for ESM Signal Processing", DREO Report No. 942, December, 1985.

[2] A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley Publishing Company, Don Mills, 1983 .

[3] P. Klahr, W. Faught, and G. Martins, "Rule-Oriented Simulation", Proc. International Conference on Cybernetics and Society, Boston, October, 1980, pp. 350-354.

[4] P. Klahr, D. McArthur, S. Narain, and E. Best, "SWIRL: Simulating Warfare in the ROSS Language", Rand Report N-1885-AF, Sept. 1982.

[5] D.A. Thomas, W.R. Lalonde and John Pugh, "A Multitasking/Multiprocessing Smalltalk", SCS-TR-92, School of Computer Science, Carleton University, Ottawa, May, 1986.

[6] James F. Cunningham, et.al., "Modelling an EW System Using an Object Oriented Approach", Proc. Expert Systems in Government Symposium, McLean, Virginia, October, 1985, pp. 406-413.

[7] W. Morven Gentleman, "Using the Harmony Operating System", National Research Council of Canada Report No. 24685, May 1985.

[8] D.A. Thomas, "Object Oriented Design of Multiprocessor Software Using the Harmony Operating System", Dy-4 Systems Design Note, 1987.

[9] Brad J. Cox, "Object Oriented Programming: An Evolutionary Approach", Addison-Wesley Publishing Company, Don Mills, 1986.

[10] Alan Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", ACM Transactions on Programming Languages and Systems, Vol. 3, No.4, October 1981.

[11] Robert A. Duisberg, "Animated Graphical Interfaces Using Temporal Constraints", Technical Report No. CR-86-05, Computer Research Laboratory, Tektronix Industries, Beaverton, Oregon, 1986.

[12] Randall B. Smith, "The Alternate Reality Kit, An Animated Environment for Creating Interactive Simulations", Proc. IEEE Computer Society Workshop on Visual Languages, Dallas, June, 1986.

[13] J.A. Altoft and Brian M. Barry, "An Object-Oriented Emitter Database for EW Applications", DREO Technical Note 87-25, September, 1987.

[14] Wilf R. Lalonde, Private Communication, October 1986.

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1 ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.) DEFENCE RESEARCH ESTABLISHMENT OTTAWA DEPARTMENT OF NATIONAL DEFENCE SHIRLEY BAY, OTTAWA, ONTARIO K1A 0Z4 CANADA	2 SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable) UNCLASSIFIED

3 TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title.)

OBJECT-ORIENTED SIMULATION OF EW SYSTEMS (U)

4 AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

BARRY, BRIAN M.

5 DATE OF PUBLICATION (month and year of publication of document) DECEMBER 1987	6a NO OF PAGES (total containing information. Include Annexes, Appendices, etc.) 61	6b NO. OF REFS (total cited in document) 14

6 DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

DREO TECHNICAL NOTE

8 SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

9a PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant) 011LB13	9b. CONTRACT NO (if appropriate, the applicable number under which the document was written)
10a ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.) DREO TECHNICAL NOTE 87-31	10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor) DRP 87-368

11 DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(XX) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12 DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM

DCD03 2/06/87

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

(U) Simulations of complex EW systems are difficult to build and virtually impossible to thoroughly validate. As a consequence, most EW systems engineers tend to regard results derived from simulations as suspect, preferring to relay instead on laboratory testing and field trials for performance evaluations. We suggest that the real problem may be that traditional simulations do not provide the kind of modelling and analysis tools which the systems engineer really needs. In this paper a prototype for a new kind of EW simulation environment which supports an object-oriented approach to modelling and simulation is described. We will provide some background information on object-oriented programming, describe the software architecture of the simulation environment, and discuss several examples which illustrate its use.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Simulation
Object-Oriented Programming
EW Systems Analysis
Validation
Smalltalk